Advanced Deployment

Architectural Composition

Saverio Giallorenzo | sgiallor@cs.unibo.it

Previously on **Jolie**

```
include "console.iol"
include "myService.iol"
```

Architectural Composition

A composition technique based on deployment abstractions.

Architectural Composition

A composition technique based on deployment abstractions.

E.g., a **service may execute other sub-services** in the same execution engine for performances and resource control.

Architectural Composition

A composition technique based on deployment abstractions.

E.g., we want proxies, redirections, and specific hierarchies to create a particular **topology** of a **Service-Oriented Architecture (SOA)**

Architectural Composition - Embedding

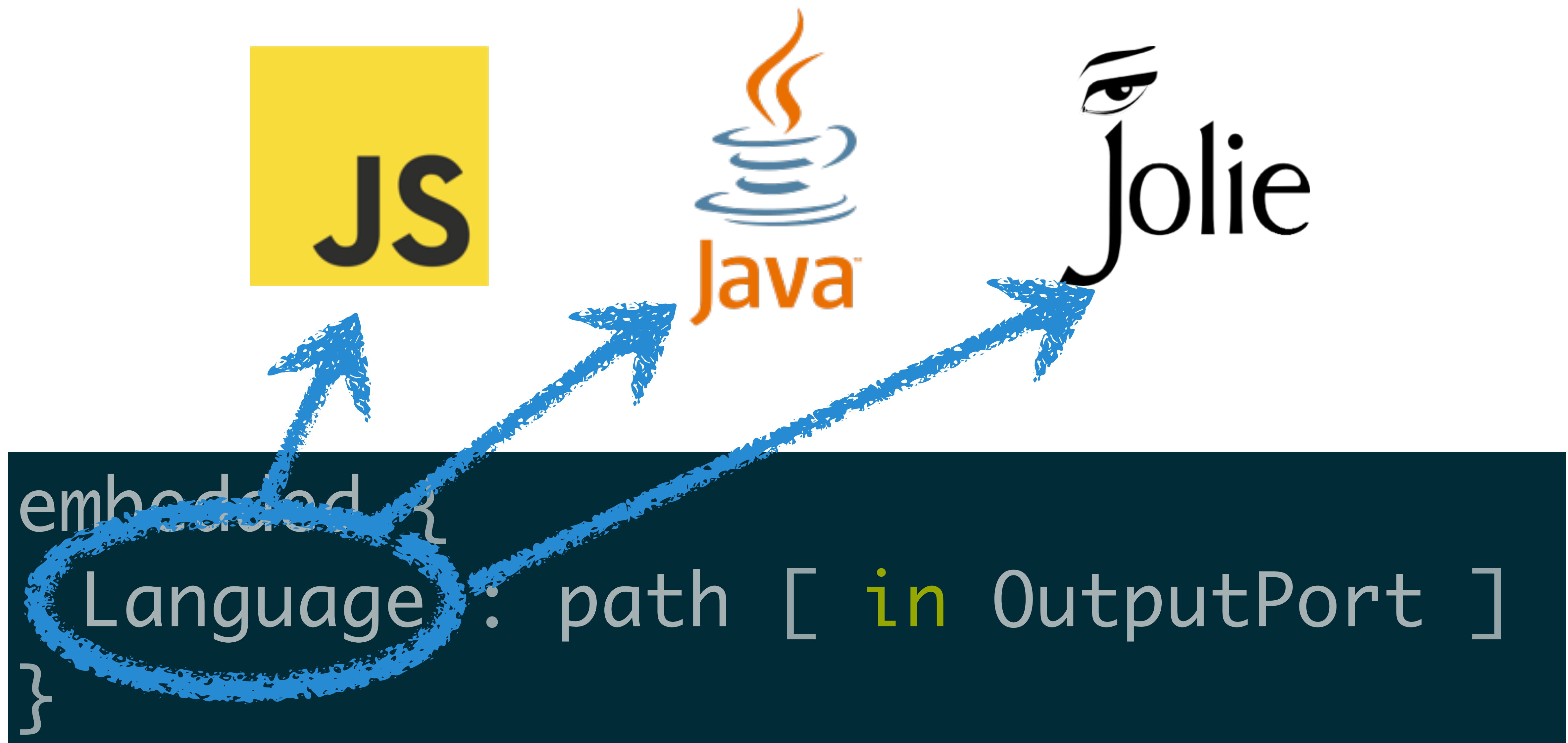
Embedding executes multiple services in the same virtual machine.

The **embedder** embeds an **embedded** service by targeting it with the **embedded** primitive.

```
embedded {
  Language : path [ in OutputPort ]
}
```

Architectural Composition - Embedding

Embedding executes multiple services in the same virtual machine.



Architectural Composition - Embedding Jolie Services

```
include "twiceInterface.iol"

inputPort LocalIn {
   Location: "local"
   Interfaces: TwiceInterface
}

main
{
 twice( number )( result ) {
  result = number * 2
 }
}
```

twice_service.ol

```
include "twiceInterface.iol"
include "console.iol"

outputPort TW {
 Interfaces: TwiceInterface
}

embedded {
 Jolie: "twice_service.ol"
  in TW
}

main
{
  twice@TW( 5 )( response );
  println@Console( response )()
}
```

Architectural Composition - Embedding Java Services

When embedding a Java service:

- the **path URL** must unambiguously identify a Java class;
- the **class** must be **in the Java classpath** of the Jolie interpreter;
- the class must **extend** the **abstract class** jolie.runtime.JavaService, offered by the Jolie library (jolie.jar) inside the Jolie installation folder.

Architectural Composition - Embedding Java Services

Many services of the Jolie standard library (like **Console**) are Java services.

Each public method of the Java Service **is an input operation** invokable by the embedder.

If the output of the method is:

- **void**, it will be a **one-way** operation
- **non-void,** it will be a **request-response** operation

(`@RequestResponse` annotation overrides this for void-returning operations).

Architectural Composition - Embedding Java Services

MyConsole.java

```java
package example;
import jolie.runtime.JavaService;

public class MyConsole extends JavaService {

  public void println( String s  ){
    System.out.println( s )
  }
}
```

Architectural Composition - Embedding Java Services

MyConsole.java

We have to compile the class into a **.jar library**.

- `javac -cp $JOLIE_HOME/jolie.jar MyConsole.java;`
- `jar cvf example.jar example/MyConsole.class`
- then either:
  - `jolie -l /path/to/example.jar myService.ol`
  - `jolie myService.ol` (with example.jar in the folder of execution of myService.ol or in the folder "javaService" under $JOLIE_HOME)

```
package example;
import
jolie.runtime.JavaService;

public class MyConsole
extends JavaService {

 public void
  println( String s  ){
   System.out.println( s )
 }
}
```

Architectural Composition - Embedding Java Services

Value and **ValueVector** objects to handle a custom-typed structures in JavaServices.

```
import jolie.runtime.JavaService;
import jolie.runtime.Value;
import jolie.runtime.ValueVector;
//...
     Value pinco = Value.create();
     pinco.getNewChild("name").setValue("Pinco");
     pinco.getNewChild("surname").setValue("Pallino");

     Value paperino = Value.create();
     paperino.getNewChild("name").setValue("Paolino");
     paperino.getNewChild("surname").setValue("Paperino");

     ValueVector response = new ValueVector();
     response.set( 0, pinco );
     response.set( 1, paperino );
//...
```

Architectural Composition - Embedding JavaScript

Embedding a JavaScript Service enables to use both the **JavaScript** language **and Java** methods by importing their classes.

```
importClass( java.lang.System );
importClass( java.lang.Integer );

function twice( request )
{
   var number = request.getFirstChild("number").intValue();
   System.out.println( "'Twice' request for number: " + number );
   return Integer.parseInt(number + number);
}
```

Architectural Composition - Embedding JavaScript

```
importClass( java.lang.System );
importClass( java.lang.Integer );

function twice( request )
{
 var number  =
request.getFirstChild("number").intValue();
  System.out.println( "'Twice' request for
   number: " + number );
  return Integer.parseInt(number + number);
}
```

TwiceService.js

```
include "console.iol"

outputPort TwiceService {
Interfaces: TwiceInterface
}

embedded {
JavaScript:
   "TwiceService.js" in TwiceService
}

main
{
   request.number = 5;
   twice@TwiceService( request )
    ( response );
   println@Console( “Response: " +
    response )()
}
```

Architectural Composition - Dynamic Embedding

Dynamic embedding associates a **unique embedded instance** (session) to the embedder.

```
include "runtime.iol"

outputPort CounterService{
   Interfaces: CounterInterface
}

execution{ concurrent }

main
{
   startNewCounter();

   embedInfo.type = "Jolie";
   embedInfo.filepath = "CounterService.ol";

   loadEmbeddedService@Runtime( embedInfo )( CounterService.location );

   start@CounterService();
   //...
}
```

Include the runtime.iol library

Embedding settings

loadEmbeddedService returns the (local) location of the embedded service

Architectural Composition - Aggregation

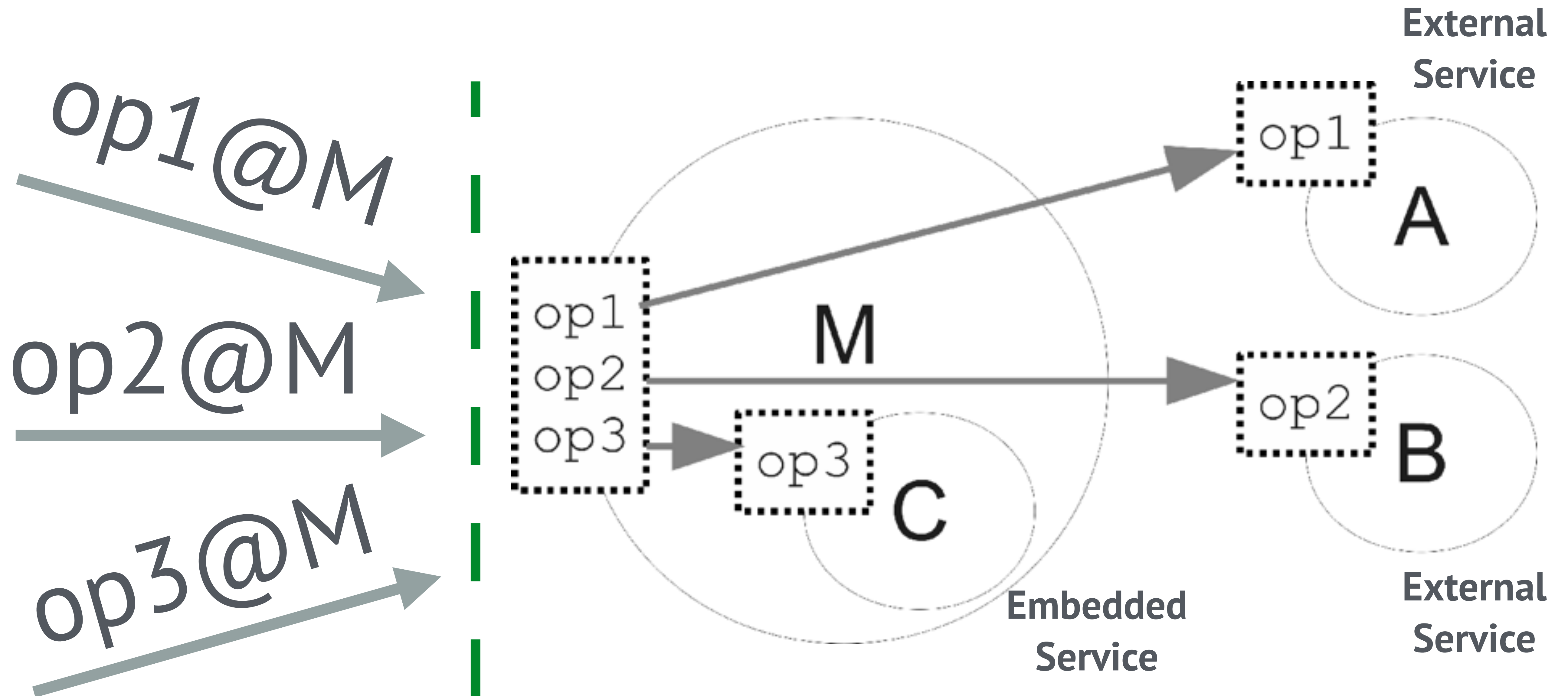
A generalisation of proxies. Allows a service to expose operations it does not implement.

It **delegates** them to the services it **aggregates.**

```
inputPort id {
  Location: URI
  Protocol: p
  Interfaces: iface_1, ..., iface_n
  [ Aggregates: outputPort_1, outputPort_2, ... ]
}
```

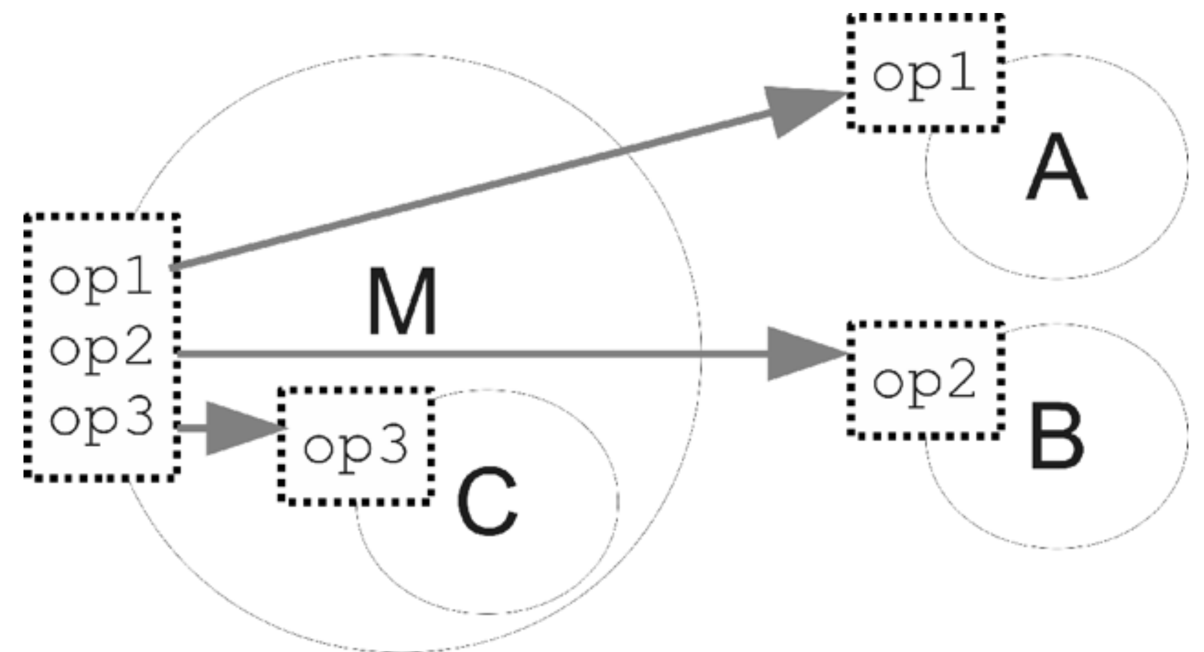
Attention: when aggregating outputPorts the **inputPort must declare** their interfaces to expose their operations.

Architectural Composition - Aggregation



Architectural Composition - Aggregation

```
outputPort A {
   Location: "socket://someurlA.com:80/"
   Protocol: soap
   Interfaces: InterfaceA
}

outputPort B {
   Location: "socket://someurlB.com:80/"
   Protocol: xmlrpc
   Interfaces: InterfaceB
}

outputPort C {
   Interfaces: InterfaceC
}

embedded {
   Java: "example.serviceC" in C
}

inputPort M {
   Location: "socket://urlM.com:8000/"
   Protocol: sodep
   Aggregates: A, B, C
}
```

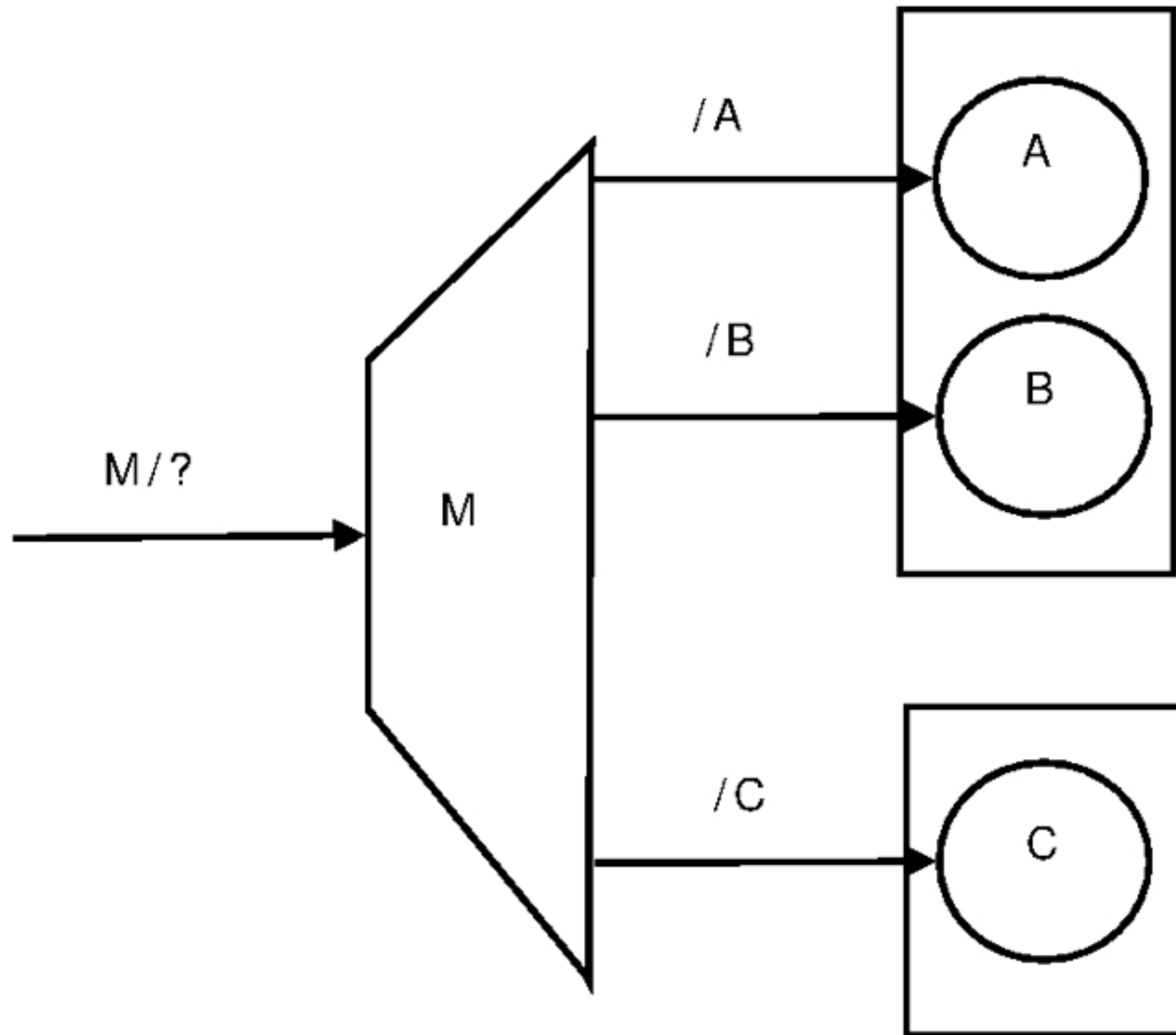


Architectural Composition - Redirection

Allows the creation of a **master service** acting as a single communication endpoint to multiple service, called **resources**.

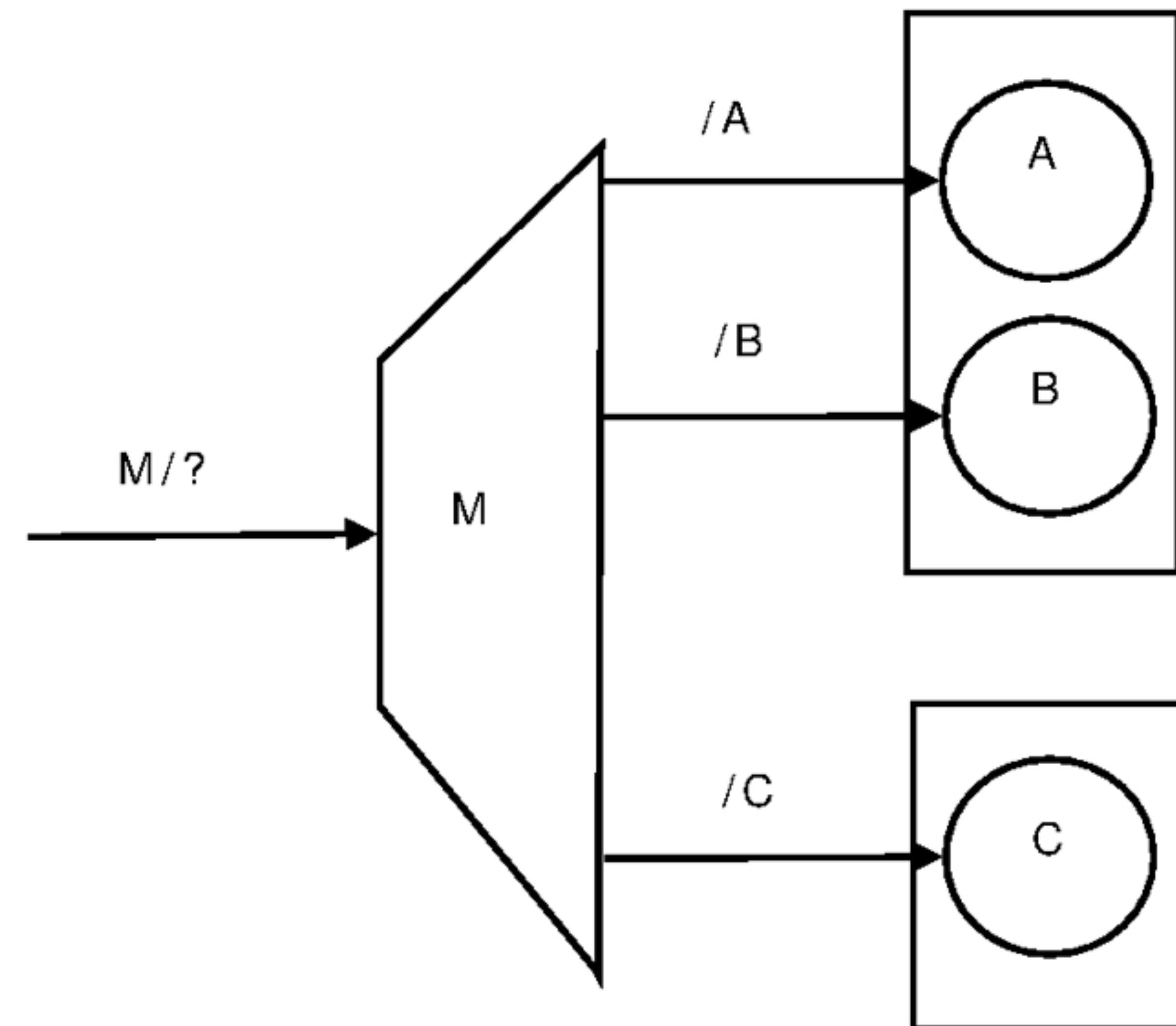
```
inputPort id {
  Location: URI
  Protocol: p
  Redirects:
    res1 => OutputPortToService1,
    // …
    resN => OutputPortToServiceN
}
```

Architectural Composition - Redirection

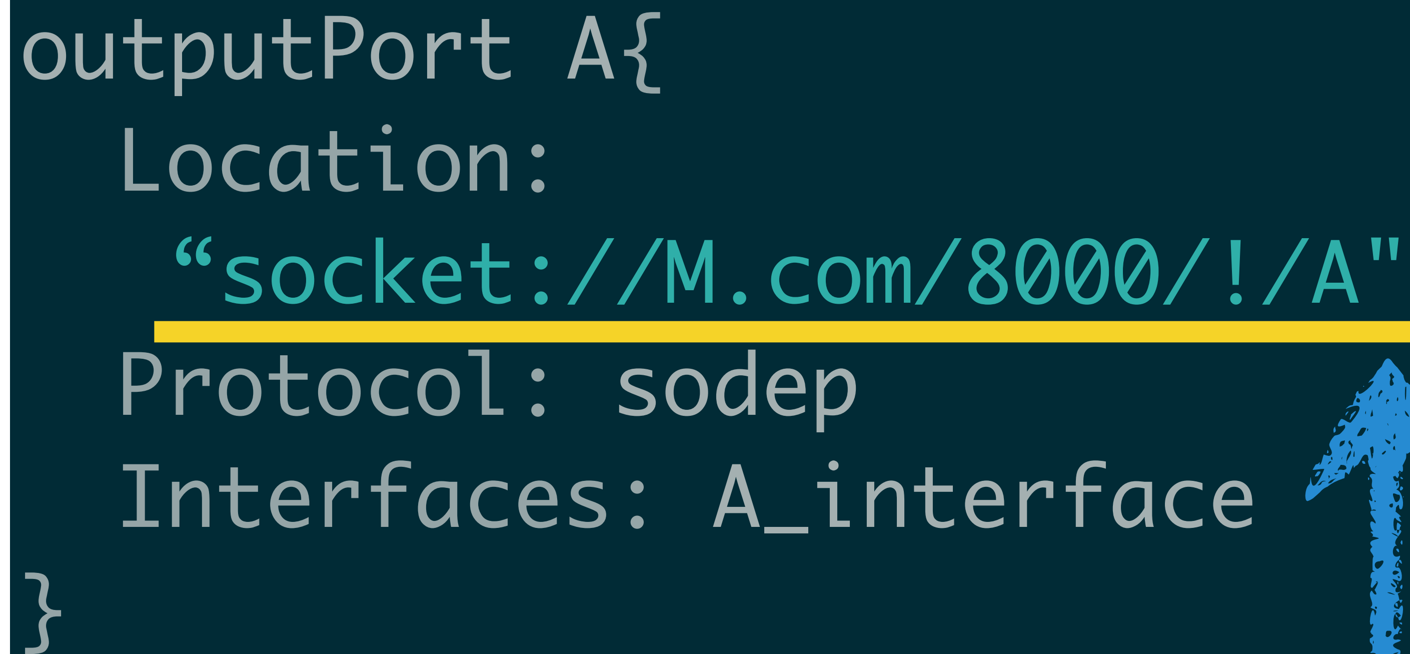


Architectural Composition - Redirection

```
outputPort ServiceA {
   Location: “socket://A.com/"
   Protocol: soap
   Interfaces: A_interface
}

outputPort ServiceB {
   Location: “socket://B.com/"
   Protocol: sodep
   Interfaces: B_interface
}

outputPort ServiceC {
   Location: “socket://C.com/"
   Protocol: http
   Interfaces: C_interface
}

inputPort M {
   Location: “socket://M.com:8000/"
   Protocol: sodep
   Redirects:
      A => ServiceA,
      B => ServiceB,
      C => ServiceC
}
```



Architectural Composition - Redirection



The address of a resource declared by a master service is **the location of the master** followed by the resource name separator **!/** and the **name of the resource.**

```
outputPort ServiceA {
    Location: “socket://A.com/"
    Protocol: soap
    Interfaces: A_interface
}

outputPort ServiceB {
    Location: “socket://B.com/"
    Protocol: sodep
    Interfaces: B_interface
}

outputPort ServiceC {
    Location: “socket://C.com/"
    Protocol: http
    Interfaces: C_interface
}

inputPort M {
    Location: “socket://M.com:8000/"
    Protocol: sodep
    Redirects:
        A => ServiceA,
        B => ServiceB,
        C => ServiceC
}
```

Architectural Composition - Dynamic Redirection

Redirection can be changed dynamically changing the binding of its outputPort.

```
outputPort SumService {
 Interfaces: SumInterface
}

inputPort MyService {
 Location: "socket://localhost:2000/"
 OneWay: balance( void )
 // no need to declare SumInterface
 Protocol: sodep
 Redirects: Sum => SumService
}

main
{
 sumService[0].location = “socket://S1.com:2001";
 sumService[0].protocol = "soap";
 sumService[1].location = “socket://S2.com:2002";
 sumService[1].protocol = "sodep";
 SumService << sumService[ i ];
 while( true ){
  balance();
  SumService << sumService[ i++%2 ]
 }
}
```