

**SIXTH FRAMEWORK PROGRAMME
PRIORITY 2
“Information Society Technologies”**

Project acronym: RUNES

Project full title: Reconfigurable Ubiquitous Networked Embedded Systems

Proposal/Contract no.: IST-004536-RUNES

**D5.2.2
RUNES Prototype Mobile Middleware with Generic
Components**

Project Document Number: RUNES/D5.2.2/ (official version)

Project Document Date: 10/11/2005

Workpackage Contributing to the Project Document: WP5

Deliverable Type and Security: R¹-PU

Author(s): Cecilia Mascolo, Stefanos Zachariadis (University College London)

Gian Pietro Picco, Paolo Costa, Luca Mottola (Politecnico di Milano)

Geoff Coulson, Nelly Becomo, Paul Okanda, Thirunavukkarasu Sivaharan (University of Lancaster)

¹Type: P - Prototype, R - Report, D - Demonstrator, O - Other

Deliverable 5.2.2

Prototype Mobile Middleware with Generic Components

Paolo Costa^{*}, Geoff Coulson[#], Cecilia Mascolo[†], Luca Mottola^{*},
Gian Pietro Picco^{*}, Thirunavukkarasu Sivaharan[#] and Stefanos Zachariadis[†]

[#]Dept. of Computing
Lancaster University
{geoff|siva}@comp.lancs.ac.uk

[†]Dept. of Computer Science
University College London
{c.mascolo|s.zachariadis}@cs.ucl.ac.uk

^{*}Dip. di Elettronica ed Informazione
Politecnico di Milano
{costa|mottola|picco}@elet.polimi.it

1 Introduction

Modern embedded distributed systems are typically composed of a high number of heterogeneous devices running different operating systems and communicating through a variety of different network interfaces. In such scenarios, there is a clear need for a framework to support application developers in the design and programming of such complex systems. The RUNES middleware architecture aims to provide this framework in the form of a *Component-based middleware*. This particular approach allows developers to decouple functionalities and perform dynamic reconfiguration of the middleware infrastructure, thus providing the mechanisms needed to hide heterogeneity and face the dynamic scenarios typical of embedded networked distributed systems.

The RUNES middleware architecture has been defined in a previous document [2]. There, the core concepts of the RUNES component model have been described and the needed interfaces defined in terms of IDL signatures. The goal of this document is to describe how these concepts have been incarnated in an actual implementation of the RUNES architecture written in the Java language [1].

The rest of the document is organized as follows: Section 2 details on the mapping from the RUNES meta-model to the Java implementation, Section 3 describes how the needed functionalities have been realized and how the method invocations unfold in the provided implementation, Section 4 constitutes a brief tutorial to the task of writing components in this framework. Finally, Section 5 points out the work that remains to be done in the Java implementation of the RUNES middleware architecture.

2 Static View

We here describe how we mapped concepts of the RUNES middleware architecture to the Java language. In this sense, the code in the current implementation is organized in packages in order to separate the actual IDL-to-Java mapping from the realized mechanisms. In particular, the `runes.core` and `runes.core.capsuleInternals` packages contain the actual mapping, while the `runes.core.defaultImpl` and `runes.core.capsuleInternals.defaultImpl` packages contains the implementation of the needed mechanisms.

As usual, IDL *modules* and *sequences* have been mapped to a Java *packages* and *arrays* respectively. The *Attribute* structure has been realized with a final Java class providing simple methods to set and get the key-value pair stored in an *Attribute*. IDL interfaces have been mapped to Java interfaces with a one-to-one correspondence of operation signatures to Java methods. The *is-a* relation in the RUNES meta-model [2] has been realized as extension among Java interfaces. To signal errors in performing operations, exceptions are raised (see the `runes.core.exceptions` package). An overall picture of the resulting JAVA mapping is given in Figure 1.

As for the actual implementation of the necessary mechanisms, the Java interfaces *Component*, *ComponentFramework* and *Connector* have been implemented in the abstract Java classes *BaseComponent*, *BaseComponentFramework* and *BaseConnector*. These will have to be subclassed by components written by application developers. To avoid redundancies and minimize the code a component programmer has to write,

Finally, simple Receptacles and Interfaces for point-to-point connections among Components are provided in `SimpleInterface` and `SingleReceptacle`. Notice that application components are not allowed to declare or use Receptacles directly. For this purpose, protected methods are provided in `BaseComponent`. An overall picture of the implementation is provided in Figure 2.

3 Dynamic View

The basic operation to be performed in the RUNES middleware architecture are `ComponentType` loading and Component instantiation. In this sense, let us remember that a `bind()` operation performed to connect two Components ultimately still consists in a Component instantiation, with this Component being a Connector representing the connection. We here highlight how these mechanisms have been realized by showing the sequence of method invocations needed to accomplish these tasks.

The processing usually starts from the invocation of a method on an object instantiated from the `DefaultCapsule` class. Figure 3 shows the sequence of method invocations necessary to load a `ComponentType`. There, we assumed that the load operation is requested by some external Component bind to the Capsule via a Connector. The actual loading of the `ComponentType` is performed through a chain of delegations from the Capsule up to the Pattern. The sequence of operation needed to instantiate a Component is instead shown in Figure 4. Apart from the actual instantiation, the `DefaultCapsule` also takes care of retrieving basic information from the just created Component, i.e., the list of its Interfaces and Receptacles, and put them in the registry.

4 A Sample Application

In this section we provide a brief introduction to the task of writing components for the Java RUNES middleware architecture through a sample application. Our goal is to write a simple component-based calculator able to perform additions and multiplications. These operations will be performed by connecting the component representing the Calculator to two components implementing addition and multiplication, respectively. The final architecture we want to obtain is represented in Figure 5. The receptacle of the Calculator component is meant to be used by another application component willing to use the services provided by Calculator.

4.1 Writing Interfaces and Components

Let us start from the Interfaces of Components. In this simple example, we just need three Interfaces, one for each component we plan to implement. In particular, we need an `IAdder` interface with a single operation used to add two integers. Then, we need an `IMultiplier` interface with a single operation used to multiply two integers. Finally, an `ICalculator` interface is needed with two operations, one for adding and one for multiplying two integers. Clearly, the former will be mapped to the corresponding method in the `IAdder` interface, whereas the latter will be mapped to the corresponding method in the `IMultiplier` interface. The three interfaces are reported in Figure 6. Notice that all of them have to extend the basic RUNES interface `Interface`.

Let us point out that splitting the Calculator into multiple Components, where each of them performs a particular operation, allows one to transparently and dynamically substitute the Component in charge of providing a given operation without the rest of the system being aware of this. In fact, all the Calculator Component needs to know about the components implementing addition and multiplication are their interfaces. In other words, the Calculator component depends only on the operation signatures in the `IAdder` and `IMultiplier` interfaces, not on the way these operation are realized. Therefore, the Components behind these interfaces can be changed transparently, even at runtime.

After having defined an Interface, we have to implement at least a Component implementing it. A simple Component implementing the `IAdder` interface is shown in Figure 7. Here, the addition is performed using the usual Java arithmetic operators. However, once an interface is defined, the developer can freely decide how to implement a given operation. Furthermore, notice that every time a RUNES Component is implemented, it has to be written by extending the `BaseComponent` class in order to be recognized as a RUNES Component by the rest of the framework. Apart from this, the developer has to write the actual code for all the methods defined in the Interface(s) the Component is implementing. In addition, the `construct()` and `destroy()` methods have to be implemented to perform all the necessary operations at component creation and destruction time, respectively. As in the `SimpleAdder` component there is no setup or clean-up to perform, these two methods simply print a message on the standard output.

An implementation for the Calculator component is reported in Figure 8. In this case, the `construct()` method is responsible for creating the two *single receptacles* the Component needs. This is accomplished with the

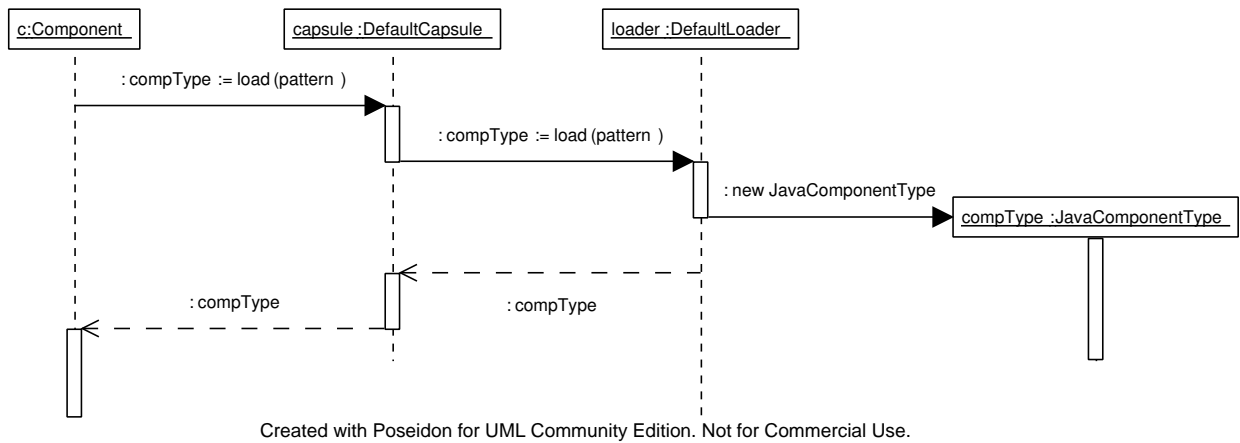


Figure 3: Sequence of method invocations performed to load a ComponentType starting from a Pattern.

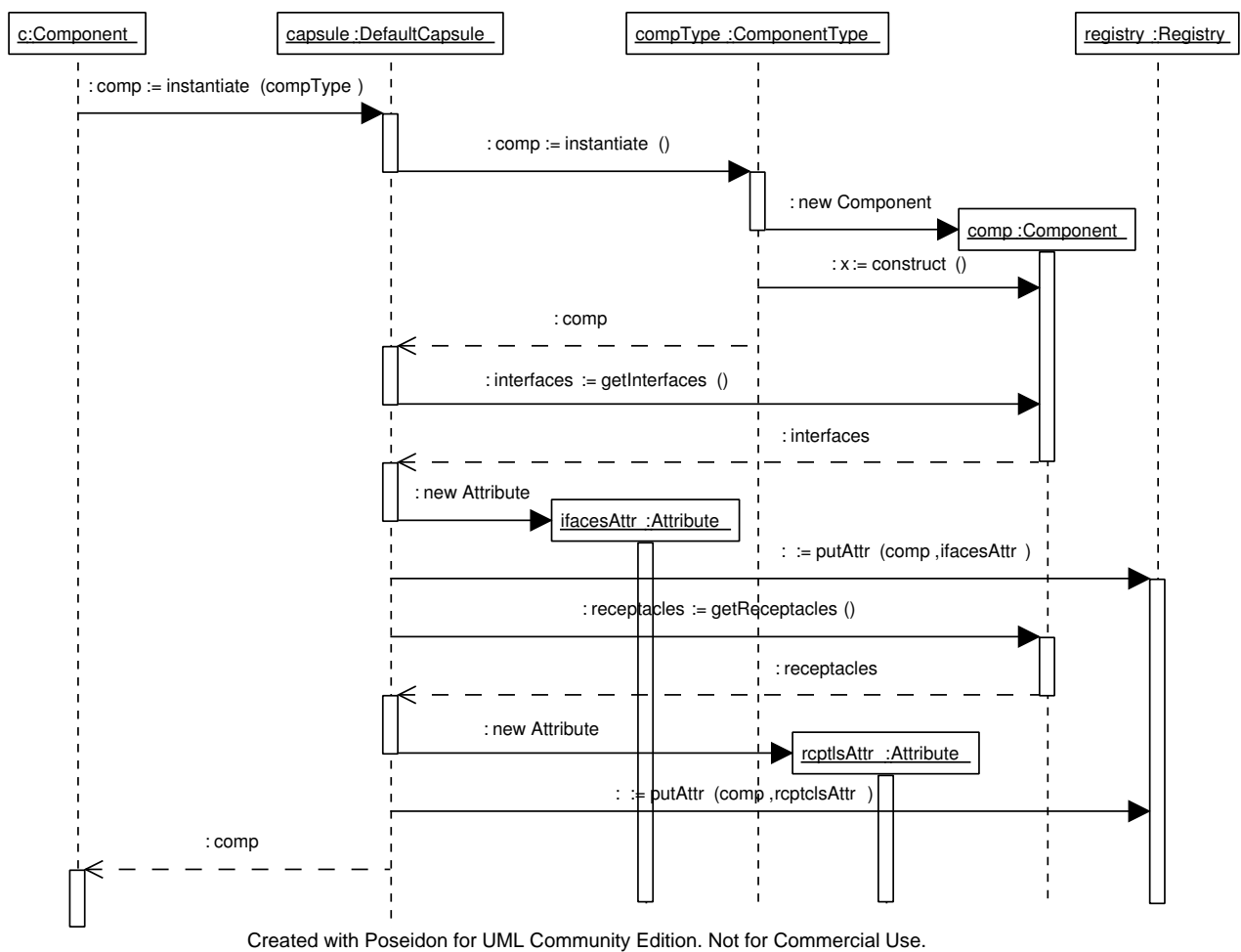


Figure 4: Sequence of method invocations performed to instantiate a Component from a Component.

by means of the `getSingleConnectedInterface()` method, that returns a reference to the Interface connected to this Receptacle. Notice that there are no guarantees that the Receptacle is actually connected to a given Interface. For instance, a Receptacle might turn out to be disconnected when the connected Component have been destroyed. For

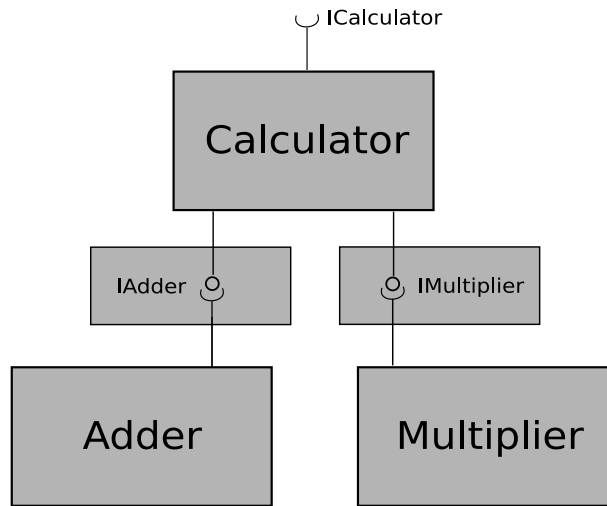


Figure 5: Architecture of a simple component-based Calculator. Notice that connections between Components are also represented as Components. In particular, these will be Connector Components.

```

public interface IAdder extends Interface {
    public int add(int x, int y);
}

public interface IMultiplier extends Interface {
    public int multiply(int x, int y);
}

public interface ICalculator extends Interface{
    public int add(int x, int y);
    public int multiply(int x, int y);
}

```

Figure 6: Interfaces for a component-based Calculator.

```

public class SimpleAdder extends BaseComponent implements IAdder {

    public void construct() throws ComponentException {
        System.out.println("Adder component instantiated");
    }

    public void destroy() throws ComponentException {
        System.out.println("Adder component destroyed");
    }

    public int add(int x, int y) {
        return x+y;
    }
}

```

Figure 7: A Component implementing the IAdder Interface.

this reason, every time a `getSingleConnectedInterface()` method is invoked, one has to explicitly manage the exceptions possibly raised. Moreover, let us point out that the `getSingleConnectedInterface()` method has to be invoked systematically every time we want to use a `SingleReceptacle`. Indeed, if one would simply invoke it at Component creation time and store the reference to the Interface internally, then it might be the case that the reference gets not updated in case the connected Component is destroyed or changed.

4.2 Writing Connectors

Once we created a Component for each of three Interfaces we defined, we have to wire the Components together in order to perform some useful computation. To this end, we need to implement a Connector Component for each of the Interfaces we defined. A Connector is a particular Component meant to represent the connection between two standard Components. It can either simply pass the operation to the connected Component without further intervention or perform some more complex operation before passing the operation to the connected Component, e.g. to monitor communication between the connected parties or to implement a distributed connection among Components residing on different hosts. A simple Connector for the `IAdder` interface is shown in Figure 9. Clearly, this is the most simple Connec-

```

public class Calculator extends BaseComponent implements ICalculator {

    public void construct() throws ComponentException {
        createSingleReceptacle("runes.sampleApp.IAdder");
        createSingleReceptacle("runes.sampleApp.IMultiplier");
        System.out.println("Calculator component instantiated");
    }

    public void destroy() throws ComponentException {
        System.out.println("Calculator component destroyed");
    }

    public int add(int x, int y) {
        SingleReceptacle r = getSingleReceptacleTo("runes.sampleApp.IAdder");
        try {
            return ((IAdder) r.getSingleConnectedInterface()).add(x, y);
        } catch (ReceptacleException e) {
            if (e.getKind() == ReceptacleException.NOT_CONNECTED)
                System.err.println("Adder Component not connected!");
        }
        return 0;
    }

    public int multiply(int x, int y) {
        SingleReceptacle r = getSingleReceptacleTo("runes.sampleApp.IMultiplier");
        try {
            return ((IMultiplier) r.getSingleConnectedInterface()).multiply(x, y);
        } catch (ReceptacleException e) {
            if (e.getKind() == ReceptacleException.NOT_CONNECTED)
                System.err.println("Multiplier Component not connected!");
        }
        return 0;
    }
}

```

Figure 8: A simple implementation for the ICalculator Interface.

```

public class IAdderDefaultSingleConnector extends BaseConnector
implements IAdder {

    public void construct() throws ComponentException {
        System.out.println("IAdderDefaultSingleConnector instantiated");
    }

    public void destroy() throws ComponentException {
        System.out.println("IAdderDefaultSingleConnector destroyed");
    }

    public int add(int x, int y) {
        return ((IAdder) getConnectedInterface()).add(x, y);
    }
}

```

Figure 9: A simple implementation of a Connector for the IAdder Interface.

tor one could implement, as it simply pass the operation to the Component implementing the IAdder interface. Notice that, as the SimpleAdder Component extends the BaseComponent class, the IAdderDefaultSingleConnector Connector extends the BaseConnector class. In addition, it also implements the same Interface of the Component is meant to connect.

4.3 Wiring Components

Having implemented Components and Connectors, what remains to do is to wire everything together. To this end, the methods of the provided Capsule have to be exploited. First of all, we need to load the necessary ComponentTypes. For instance, let us consider the Calculator Component. Its ComponentType can be loaded as follows:

```
ComponentType calculatorType = capsule.load(new StringPattern( "runes.sampleApp.Calculator"));
```

With its ComponentType, we can now instantiate a Component of type Calculator as follows:

```
Component calculator = capsule.instantiate(calculatorType);
```

In the same way, we can load ComponentTypes for the SimpleAdder and SimpleMultiplier Components and then instantiate two of them. Let us call them adder and multiplier.

We are now ready to connect Components together. For instance, to connect the Calculator to a SimpleAdder, we first retrieve a reference to the Interface of the SimpleAdder Component and to the corresponding Receptacle in Calculator. Once we obtained these references, we invoke the bind() method of the Capsule and get in return a Connector component representing the binding:

```
Interface adderIf = (Interface) capsule.getAttr(adder, "INTERFACE-runes.sampleApp.IAdder").getValue();
Receptacle calculatorAdderRecept = (Receptacle) capsule.getAttr(calculator, "RECEPTACLE-runes.sampleApp.IAdder").getValue();
Connector calcToAdder = capsule.bind(adderIf, calculatorAdderRecept);
```

In case we later want to destroy this binding, we destroy the Connector component representing it as we would destroy any other Component, i.e., by means of the `destroy()` method.

5 Future work

The current implementation provides the basic mechanisms to load ComponentTypes, the facilities needed to instantiate, destroy and bind Components, basic registry mechanisms, and basic support for writing ComponentFrameworks[2]. However, it does not yet have support for *reflective extensions*, as well as it does not yet provide the mechanisms to define and enforce Constraints on ComponentFrameworks. These features are currently under investigation and will be incorporated in later versions of the Java RUNES middleware architecture.

References

- [1] Java language specification. java.sun.com/docs/books/jls/.
- [2] C. Mascolo, S. Zachariadis, G.P. Picco, P. Costa, L. Mottola, G. Blair, N. Bencomo, P. Okanda, and T. Sivaharan. D5.2.1 RUNES Middleware Architecture. *RUNES Project*.