

**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

D5.2.3

**RUNES Prototype Mobile Middleware with Basic
Application Specific Components**

Project Document Number: RUNES/D5.2.3/ (official version)

Project Document Date: 03/02/2006

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, Thirunavukkarasu Sivaharan (University of Lancaster)

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

Deliverable 5.2.3

Prototype Mobile Middleware with Basic Application Specific 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

The RUNES middleware architecture aims at providing a framework for the flexible and effective development of distributed applications for networked embedded systems. This framework is based on the notion of *Component*, an encapsulated unit of functionality and deployment. This particular approach allows developers to decouple functionalities and perform dynamic reconfiguration of the middleware infrastructure, thus providing the mechanisms needed to face the heterogeneity usually found in embedded, networked distributed systems.

The RUNES middleware component model has been defined in a previous document [4]. There, the core abstractions have been described and the needed interfaces defined in terms of IDL signatures. In a following document [5], we described how those concepts have been incarnated in an actual implementation of a *middleware kernel*. This kernel, written in Java [1], provides the software support needed to build applications and *middleware services* in terms of components.

In this document we show how the component model and the corresponding support runtime we developed can be effectively used to build prototype components that realize a range of distributed middleware services. We do this by focusing on components implementing advertise and discovery services, network overlay services and logical mobility functionality. Although many other kind of distributed mechanisms can be implemented, the components we describe in this document anyway provide interesting insights on the generality and flexibility of our underlying software support architecture.

To better highlight the usefulness of the distributed mechanisms we describe in this document we refer to one of the possible RUNES application scenarios, i.e., the case of monitoring and control of a road tunnel. In this scenario, a command and control center is responsible for monitoring the activities in a tunnel by means of heterogeneous sensors deployed in the tunnel itself. Communication between the sensors and the control center is realized wirelessly by means of gateway nodes, that are responsible for collecting and aggregating data before making them available to the control center.

In case a fire is detected in the tunnel, an emergency rescue team is sent to save the people trapped inside and put out the fire. The team is equipped with mobile devices (e.g., PDAs) that allow them to communicate with the sensors in the tunnel and with possible wireless devices installed on cars or worn by people blocked in the tunnel.

The rest of the document is organized as follows: Section 2 describes the advertising and discovery services, Section 3 illustrates the overlay services component framework and discuss an example plug-in, while Section 4 shows the basics of the logical mobility functionality.

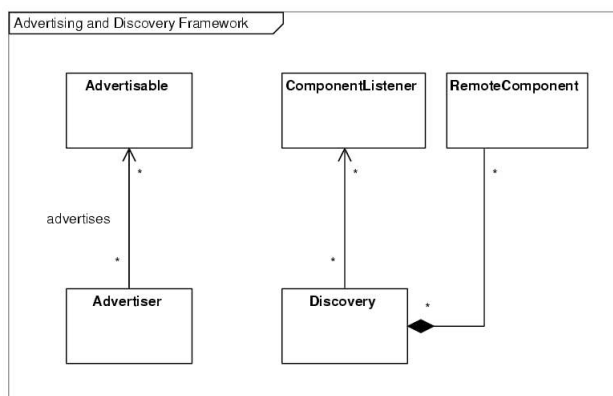


Figure 1: The RUNES Advertising and Discovery Framework.

2 Advertising and Discovery Services

One of the pivotal requirements of ubiquitous computing is the ability to discover devices and services that are offered in the environment. As a ubiquitous computing device is also part of the environment itself, it is equally important that it too is able to advertise its presence and any services it may offer. The devices that we are targeting potentially connect to different types of networks, either concurrently or at different times, with different hardware. Moreover, there are many different protocols for advertising and discovery. Imposing a particular mechanism can hinder interoperability with other systems, making assumptions about the network, the hosts and even the environment, which may be violated at some stage or not be optimal in a future setting—something which is likely to happen, given the dynamic characteristics of the environment. In the case of the RUNES middleware, where every aspect of the system is represented as a component, advertising and discovery refers to the advertising and discovery of components. As such, the RUNES advertising and discovery framework (adapted from SATIN [9]), allows dynamic advertising and discovery for components realized by different component frameworks. By using RUNES interfaces to specify a set of principles that components that wish to be advertised must adhere to, it decouples implementation from the advertising message and the representation of remote services.

2.1 Application Scenarios

In the emergency scenario we described in Section 1, as the rescue team is not aware of the configuration of the network and the measurements offered by devices installed in the tunnel or worn by people, the advertising and discovery framework can be used to offer an up to date image of which devices are available and what services they offer.

2.2 Design and Implementation

The framework is represented as a collection of interfaces plus a data structure, as shown in Figure 1. Components that wish to advertise their presence in the environment implement the `Advertisable` interface, which has a function that enables the component to export a message that can be used for advertising. Plug-in advertising “strategies” (e.g. based on uPnP, SLP etc) then implement the `Advertiser` interface. These are responsible for taking the messages of advertisable components and, after potentially transforming them into other formats, advertising them. Advertiser components allow advertisable components to register for advertising. There can be any number of Advertiser components installed.

Similarly, discovery “strategies”, which allow clients to reason about the advertisable components that have been found remotely, are encapsulated as `Discovery` components, which implement the `Discovery` interface. Discovery components use the `RemoteComponent` data structure to represent components that have been found remotely. Moreover, Discovery components allow for the registration of components that implements the `ComponentListener` interface; these components will be notified when new components have been discovered. There can be any number of discovery components available at a RUNES middleware instance.

Currently, we have reified the framework in Java (Micro Edition) using two different strategies: IP Multicast, and Centralised Publish Subscribe. We are investigating the integration with other distributed Publish Subscribe services, e.g., the one that can be implemented on top of a tree overlay as described in the following.

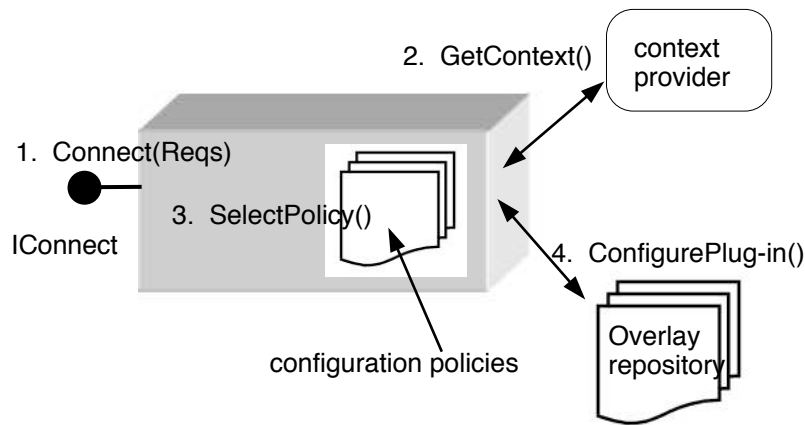


Figure 2: The overlay framework.

3 Overlay Services Component Framework

The overlay services component framework plays the crucial role of exporting underlying middleware-level communications services to other applications. A middleware platform that offers only a single overlay service plug-in, cannot cope with the diversity of requirements imposed by the various RUNES applications. Instead, a comprehensive solution needs to provide a wide range of overlay service plug-ins such as subject-based publish-subscribe overlays, content-based publish-subscribe overlays [3] and application level multicast overlays. To address this requirement, the overlay framework accommodates and configures different plug-in overlay services depending on application requirements and operational environmental context.

3.1 Application Scenarios

As operation in the road tunnel does not exhibit emergency conditions, network overlays might be required to allow for communication between the control center and the gateway nodes. This kind of overlay does not need to address highly dynamic network topologies, as gateway nodes are not envisioned to move. On the other hand, when a fire is detected, the firefighters equipped with PDAs, the gateway nodes and the sensor nodes form a mobile ad-hoc network. Therefore, there is a clear need for network overlays able to tolerate dynamic reconfigurations. Crucially, these examples illustrate the requirement for overlay services suited for both fixed networks and mobile ad-hoc networks. The overlay framework configures the appropriate overlay plug-in depending on the application requirements and operational network type.

3.2 Design and Implementation

The selection, configuration and the use of overlay plug-in components by application developers should be as straightforward as possible, and their management should be based on an (optional) declarative specification of the desired behavior by the application programmer. The application developer specifies the required overlay type and name-value pairs associated with the overlay type via the standard interface exposed by the overlay framework. The framework then automatically configures the appropriate overlay plug-in and exposes overlay plug-in interface to the application.

The *configurability* of the framework is important to support various RUNES applications, we show how different overlay types can be automatically configured in a way that is appropriate to different environmental conditions. The various overlay plug-ins needs to be first registered with the framework. We describe one such overlay plug-in later. The framework can be populated with various overlay plug-ins in the future as deemed required by different RUNES applications. The overlay framework contains a repository of XML based configuration policy files. The policy files contains rules to select the appropriate overlay plug-in based upon the overlay type and the set of name-value pairs provided by the application developer and the current environmental context. The name-value pairs deemed relevant for the given overlay type are designated by the overlay type developer. As such, the above process relies on ontology of name-values. The set of name-value pairs submitted by the application developer adheres to such domain specific ontology. The framework also contains a repository of XML based overlay plug-ins components configuration script files. Each overlay plug-in has its corresponding XML based component configuration script file. As such, when a new



Figure 3: The TOM Component with its Interfaces and Receptacles.

overlay plug-in is registered with the framework the corresponding component configuration script file is added to the repository of configuration script files. The component configuration script files specifies the list of components to be loaded, instantiated and binded to incarnate the given overlay plug-in.

The above process is illustrated in Figure 2. First connect call is made on the interface `ICoconnect` by the application on the overlay framework. Then the framework picks the overlay type that exports required API, and retrieves from the context provider the set of contextual name-value pairs that are relevant to the type of the overlay. The XML based configuration policy repository is then referred, to select the appropriate overlay plug-in component configuration script, on the basis of the predicates provided by the application and name-value pairs retrieved from the context provider. Note that the configuration policy files are added to the framework in consultation with the application developer, such that the application developer can drive the preferred configuration policy. Once the configuration script file is selected the script loads the components, instantiates and binds them to incarnate the overlay plug-in and exports the interface to the application.

The road tunnel fire scenario requires an overlay plug-in suited for mobile ad-hoc networks, such as the mobile publish-subscribe overlay plug-in which follows next.

3.2.1 An Example Overlay Plug-In

TOM (Tree Overlay Manager) is a RUNES component intended to be run as an overlay plug-in in the Overlay Services CF. Its goal is to build and maintain an application level tree-shaped overlay in MANET [2] environment, so that applicative components can route point-to-multipoint messages on top of a dynamic topology. The overlay maintenance and message transport tasks are fully decoupled to enable run-time reconfiguration as well as deployment on different hardware platforms. The necessary procedures for maintaining the interconnection topology are derived from the MAODV protocol [8, 7] with adaptations and modifications. For more information on the particular mechanisms and algorithms implemented, the reader can refer to [6].

Operation name	Description
<code>void openLink(String)</code>	The overlay manager invokes this operation when it finds a new connection in response to a previous link breakages or in case a new neighbor is acquired. It should perform all the necessary operations to set up a new logical link between neighboring nodes.
<code>String getURL()</code>	The operation should return the transport-level URL of the local host. This string will be used by other neighboring nodes for setting up a new logical link when they invoke the <code>openLink()</code> method.
<code>String getId()</code>	The operation must return a unique identifier for the local host.
<code>Collection getNeighborIds()</code>	The operation must return the set of ids of neighboring nodes as a Collection of Strings.

Table 1: The `IOverlayTransport` interface.

As shown in Figure 3, the component exports two interfaces and needs two receptacles. Among these, `IBroadcastSend` needs a simple component able to provide 1-hop broadcast facilities for serializable objects. On the other hand, `IOverlayTransport` needs to be connected to the component realizing the actual transport layer on top of the overlay network. The signatures of the operations in these two interfaces are the ones reported in Table 1. Notice that the transport component is also in charge of generating a unique id for the local host and of managing the neighbor set.

Parameter name	Description
<code>DISCOVER_TIMEOUT</code>	Represents the maximum time the local node waits to find a new link after a link breakage. In case this timeout expires without having found a replacement link, a network partition is declared.
<code>REQUEST_RETRIES</code>	Represents the maximum number of times a node will try to find a replacement link before declaring a network partition.

Table 2: TOM core parameters.

The two interfaces exported by the component are very simple: `IMessageRcv` defines a single method `receive(Object)` used to pass to the overlay manager objects received by the underlying network layer. On the other hand, `ITreeOverlayMgr`

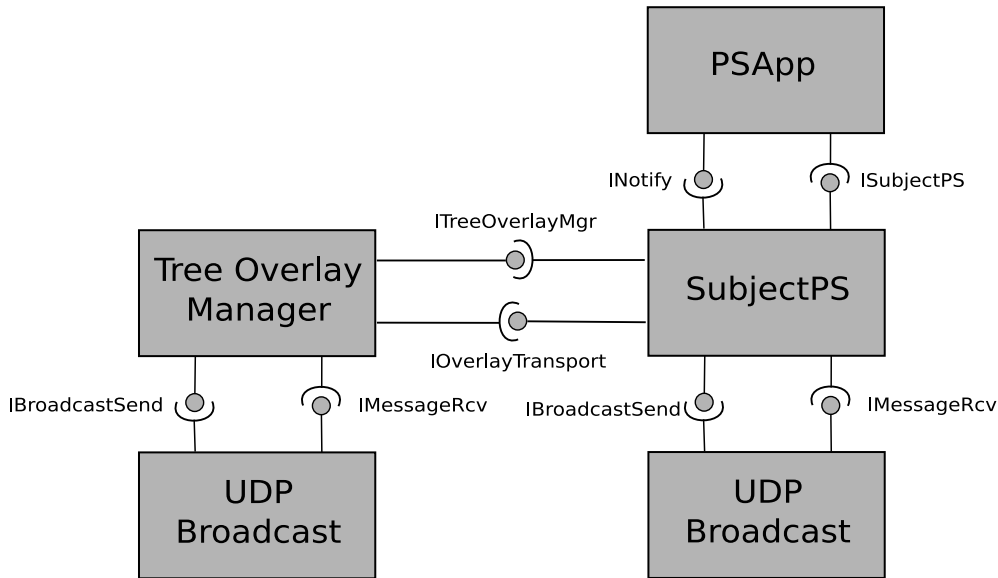


Figure 4: A possible configuration of components to implement a simple subject-based Publish-Subscribe service.

defines a single method `signalLostNeighbor(String)` used to communicate a link breakage to the overlay manager. As for the component working parameters, these can be specified in an XML file called `OverlayManagerParams`. The first time the component is instantiated, this file is automatically generated with default values. The semantics of the most important parameters is reported in Table 2.

A possible usage. A straightforward usage of TOM is to implement a simple subject-based Publish-Subscribe service [3] on top of it. In this kind of system nodes subscribe to one or more subject, where a subject corresponds to a logical channel where interested subscribers get notified when messages are published on that same channel. A simple way of implementing such a service is to arrange all nodes in a tree used to distribute published messages. To do that, we need to implement three components around the TOM, as shown in Figure 4. The `UDPBroadcast` component is used to implement 1-hop broadcast communication by means of UDP datagrams. Notice that, even if this component is here placed at the application level, in principle it could sit directly on top of a MAC layer, since no facilities for retransmissions, acknowledge or congestion control are needed. Actually, two instances of this component gets created, one for the network traffic belonging to the overlay maintenance, and one for the network traffic belonging to the Publish-Subscribe system. The `SubjectPS` component implements the subject-based Publish-Subscribe service on top of the tree overlay. To this end, it provides the TOM component with the methods specified in the `IOverlayTransport` interface. On the other side, it offers the classical APIs for a Publish-Subscribe system in the `ISubjectPS` interface, i.e. the methods `publish()`, `subscribe()` and `unsubscribe()`. Finally, the `PSApp` component implements the actual application that performs subscriptions or unsubscriptions, and publishes messages.

4 Logical Mobility Services

One of the requirements of RUNES nodes is the ability to reconfigure. Reconfiguration may refer to network reconfiguration, where the system adapts to changes in network topology, or software reconfiguration, where the system dynamically acquires functionality. The Logical Mobility Component Framework offers the functionality needed to dynamically update the codebase of a RUNES node. It allows for dynamically sending and receiving RUNES components. The components received are automatically loaded into the runtime and are ready to be instantiated and used. This CF is based on SATIN [9].

4.1 Application Scenarios

In the fire in the road tunnel scenario, the rescue team is sent into the tunnel equipped with PDAs. It is possible that as they discover the sensors that are deployed in the tunnel and the mobile devices carried by people stranded in the tunnel, they may need to reconfigure the software available on their PDAs, in order to be able to talk to them. Moreover,

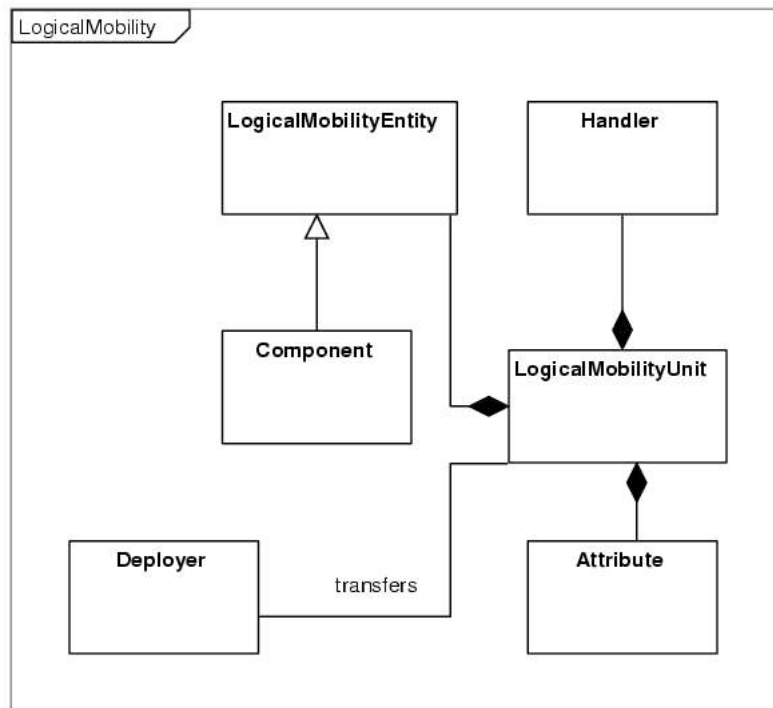


Figure 5: The RUNES LogicalMobility Framework.

the rescue team may decide to reconfigure some of the devices in the tunnel, to take specific measurements which will be useful in the rescue operation.

4.2 Design and Implementation

We define a *Logical Mobility Entity* (LME) as a generalisation of a RUNES component. The *Logical Mobility Unit* (LMU), is defined as the minimal unit of transfer in this framework. An LMU is a container, that can encapsulate components; an LMU is, in part, a composition of an arbitrary number of LMEs. The LMU provides operations that permit inspection of contents. This allows a recipient to inspect an LMU before using it. The LMU can potentially encapsulate a *Handler*. The Handler can be instantiated and the resulting object used by the recipient to deploy and manipulate the contents of the LMU. This can allow sender-customised deployment and binding.

Similarly to RUNES Components, an LMU also encapsulates a set of *attributes*, called the *properties* of the LMU. Attributes represent the metadata of the LMU. The number and type of attributes is not fixed. The properties are used to describe the LMU they are associated with. For example, logical (software) or physical (hardware) dependencies, digital signatures and even end-user textual descriptions can be expressed as attributes. As such, they can be used to express the heterogeneity of the target environment. An ontology for attribute keys and values is not defined at this stage.

We define the `Deployer` component as a RUNES component that handles sending and receiving LMUs. A Deployer can offer the following functionality:

- Serialising and deserialising the LMU
- Checking an incoming LMU for malicious code
- Checking whether a target host is trusted to send sensitive code
- Handling namespace conflicts with incoming LMUs
- Deploying an incoming LMU

Note that some of the functionality described above can be implemented in various ways (such as proof carrying code and digital signatures) and is optional. The Deployer offers an API that allows sending and receiving LMUs. If

available, it can interact with the Advertising and Discovery framework, to allow requesting and deploying Remote Components.

We have an implementation of this framework operational currently, running under the Java 2 Micro Edition platform.

4.3 Security Considerations

There are a number of security issues when dynamically sending and receiving code. The current implementation does not address any of them—when available, we will interface with the security component framework to address issues of trust and privacy.

References

- [1] Java language specification. java.sun.com/docs/books/jls/.
- [2] Mobile ad-hoc networks (manet) charter. Available at ietf.org/html.charters/manet-charter.html.
- [3] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2(35), June 2003.
- [4] 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*, 2005.
- [5] C. Mascolo, S. Zachariadis, G.P. Picco, P. Costa, L. Mottola, G. Blair, N. Bencomo, P. Okanda, and T. Sivaharan. D5.2.2 Prototype Mobile Middleware with Generic Components. *RUNES Project*, 2005.
- [6] L. Mottola, G. Cugola, and G.P. Picco. A Self-Repairing Tree Overlay to Enable Content-Based Routing in MANETs. Submitted for publication. Available at www.elet.polimi.it/upload/picco/topologyManagement.pdf, 2005.
- [7] E. Royer and C. Perkins. Multicast Ad hoc On-Demand Distance Vector (MAODV) Routing. IETF, Internet Draft. Available at www.ietf.org/internet-drafts/draft-ietf-manet-maodv-00.txt, 2000.
- [8] Elizabeth M. Royer and Charles E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *Proc. of MobiCom99*, pages 207–218, 1999.
- [9] S. Zachariadis, C. Mascolo, and W. Emmerich. SATIN: A Component Model for Mobile Self-Organisation. In *International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, October 2004. Springer.