The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems

Paolo Costa, Politecnico di Milano, Milano, Italy, *costa@elet.polimi.it*
Geoff Coulson, Lancaster University, United Kingdom, *geoff@comp.lancs.ac.uk*
Cecilia Mascolo, University College London, United Kingdom, *c.mascolo@cs.ucl.ac.uk*
Gian Pietro Picco, Politecnico di Milano, Milano, Italy, *picco@elet.polimi.it*
Stefanos Zachariadis, University College London, United Kingdom, *s.zachariadis@cs.ucl.ac.uk*

I. Introduction

Miniature computing devices are being embedded in an increasing range of objects around us including home appliances, cars, buildings, and people. Furthermore, the *networking* of such embedded environments is enabling advanced scenarios in which devices leverage off each other and exhibit autonomous and coordinated behaviour. Recent developments in wireless networking are pushing these trends even further by simplifying deployment, and enabling new applicative scenarios, as witnessed by the recent surge of interest in wireless sensor networks.

However, research into such networked embedded environments has focused almost exclusively on the development of miniaturised devices with increasingly powerful and general capabilities. As a result, the *software fabric* that ultimately makes innovative applications possible has tended to be overlooked. Instead, software is developed in an ad-hoc fashion, with little or no provision for reusable services and abstractions. Furthermore, even where attempts are made to provide such features, the wide range of devices involved in networked embedded environments inevitably leads to significant complexity in appropriately configuring, deploying, and dynamically reconfiguring the software. There is therefore a need for a *programming platform* with abstractions that are able to span the full range of heterogeneous embedded systems, and which also offers consistent mechanisms with which to configure, deploy, and dynamically reconfigure networked embedded systems software.

The work discussed in this paper is addressing the need for such a programming platform. The work is being carried out in the context of the EU-funded RUNES project (Reconfigurable, Ubiquitous, Networked Embedded Systems), which has the general general goal of developing an architecture for networked embedded systems that encompasses dedicated radio layers, networks, middleware, and specialised simulation and verification tools.

Our programming platform, which is at the heart of the RUNES architecture, takes the form of a *component-based middleware* that decouples and encapsulates the functionality provided by its various constituents behind well-defined interfaces. This decoupling not only enables one to deploy different variants of the same component (e.g., tailored to a specific device type), but also enables dynamic reconfiguration of component instances and their interconnections. This provides support for dynamic adaptation to changing conditions—a fundamental requirement in the context-aware scenarios typical of networked embedded systems. Moreover, the RUNES middleware reaches down into layers that typically belong to the network and the operating system, therefore providing a unified approach to configuration, deployment and reconfiguration at multiple levels of abstraction. Finally, the expertise of the involved partners in the related fields of mobile computing, context-aware systems, and code mobility provides the necessary knowledge and background for the design of specific components customised for particular networked embedded systems.

The rest of the paper is organised as follows. Section II introduces the component model that is the foundation of our middleware. Next, section III describes a number of key middleware services that can be built in terms of the component model. Finally, section IV offers our conclusions and plans for future work.

II. The Component Model

A. Overview

The RUNES approach to middleware provision is to build the middleware in terms of a well-defined language-independent *component model* which is supported by a minimal runtime API. The component model and its supporting runtime API are discussed in Section II-B.

The required heterogeneous realisation of the component model in various types of devices is achieved by providing different implementations of the runtime API, and by implementing components themselves in various ways. For example, on a PDA running a standard OS we might implement components as sets of Java classes or as Linux “shared objects”; whereas on a sensor mote’s microcontroller, components might be implemented simply as segments of machine code. This is possible because the component model is a *local* model:

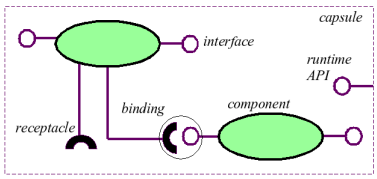

Fig. 1. Elements of the RUNES component model.

distribution is assumed to be built on top of this foundational layer.

The component model itself is complemented by two further architecture elements: *component frameworks* and *reflective meta-models*. These are discussed in Sections II-C and II-D respectively.

B. Elements of the Component Model

An outline of the component model, which is based on Lancaster's OpenCOM [1], is illustrated in Figure 1.

Components are encapsulated units of functionality and deployment that interact with other components exclusively through "interfaces" and "receptacles" (see below). *Capsules* are containing entities that offer the above-mentioned runtime API. As implied above, capsules can implemented differently on different devices—e.g. they might be implemented as a Unix or Windows process on a PDA or PC; or as a RAM chip on a sensor mote. Components can be deployed at any time during run-time, and their loading can be requested from within any component within the capsule (this is called *third-party deployment*). *Interfaces* are expressed in terms of sets of operation signatures and associated datatypes; OMG IDL is used for interface specification to give language independence. (Note, however, that this does *not* imply the overhead of CORBA-like stubs and skeletons.) Components can support multiple interfaces: this is useful in embodying separations of concern (e.g. between base functionality and component management). *Receptacles* are "required" interfaces that are used to make explicit the dependencies of a component on other components: when deploying a component into a capsule, one knows by looking at its receptacles precisely which other components must be present to satisfy its dependencies. Finally, *bindings* are associations between a single interface and a single receptacle. Like deployment, the creation of a binding is inherently third-party in nature. That is, it can be performed by any party within the capsule, not only by the first-party components that will themselves participate in the binding.

An abstract of the runtime API offered by each capsule is as follows:

```
template load(comp_type name);
comp_inst instantiate(template t);
status unload(template t);
comp_inst bind(ipnt_inst interface, ipnt_inst receptacle);
status destroy(comp_inst comp);
status putattr(ID entity, ID key, any value);
any getattr(ID entity, ID key);
```

Load() loads a named component "template" from a local repository into the capsule, and *unload()* unloads a template. Templates can be instantiated (using *instantiate()*) to yield component instances (*comp_insts*); this can be done multiple times if desired. *Bind()* is used to third-party bind a pair of "interaction points" (i.e. a receptacle and an interface). It returns a *comp_inst* that represents the binding; this, like any other component instance, can be destroyed using *destroy()*.

The *putattr()* and *getattr()* calls give access to a general purpose *registry* facility. This allows allow arbitrary <*key*, *value*> attributes to be associated with component model entities (i.e. components and interaction points). It provides the basic means by which components discover each other at runtime: to this end, *load()* and *instantiate()* register the existence of each new template or component instance under a well-known key. It is also used to obtain the interaction points associated with a component, and to associate QoS properties etc. with component model entities.

C. Component Frameworks

A *component framework* [2] is an encapsulated composition of components that addresses some focused area of functionality, and which accepts additional components as run-time "plug-ins" that somehow modify or extend the component framework's behaviour (examples are given below). Crucially, component frameworks (hereafter CFs) actively police attempts to plug-in new components according to well-defined, per-CF, it policies and *constraints* that are expressed in a language such as OCL.

The benefits of CFs are as follows: *i*), they provide an intermediate abstraction between components and whole systems and thus generally increase the understandability and maintainability of systems; *ii*) they simplify component development and assembly through design reuse and guidance to developers; and *iii*) they enable *lightweight* components (plug-ins) because plug-ins can assume shared CF-specific state and services.

Specific examples of CFs are given in Section III-A.

D. Reflective Meta-Models

The essence of "reflection" is to establish and manipulate causally-connected *meta-models* of an underlying target system [3]. Such meta-models are representations of some aspect or view of the target system, and they expose a so-called *meta-interface* through which the representation can be inspected and manipulated. The main purpose of reflection is to maintain an architectural separation of concerns between system building or *configuration* (sometimes called base-level programming), and system adaptation or *reconfiguration* (sometimes called meta-programming).

Reflection is a powerful technique, and its use should ideally be constrained to minimise programmer error. Our approach is to deploy reflective meta-models in close association with

CFs. The idea is that CFs can encapsulate meta-interfaces, and appropriately restrict access to them according to policy. Furthermore, such encapsulation can also ensure that meta-interfaces are accessed only when conditions are *safe*; for example, a CF could restrict component replacement via the *architecture meta-model* (see Section III-B) to situations in which no invocations are currently being made across interaction points owned by the "old" component. Further, a CF can define a suitable state-transfer protocol to carry-over essential state from the old component to the new one.

Specific examples of reflective meta-models are given in Section III-B.

III. Towards the RUNES Middleware

We now show how the abstract concepts offered by the component model can be instantiated as specific RUNES software elements. First, in Section III-A, we describe a number of possible RUNES CFs; then, in Section III-B, we describe some example reflective meta-models. Finally, in Section III-C, we give an example of how multiple CFs can work together.

A. Examples of RUNES CFs

The following are examples of RUNES CFs:

Local OS services. This CF exposes as plug-ins functionality (or abstractions of functionality) that is typically provided by the host operating system running, such as local scheduling, memory management, or MAC protocols. As an example, different thread scheduling polocies can be provided by different plug-ins.

Network Services. This CF supports an extensible set of plug-in network communication services and provides a uniform set of APIs to these. It accommodates both ad-hoc networking and infrastructure-based networking (in the latter the plug-ins tend to be overlays). Examples of plug-ins that might be accepted by the CF are: flooding, flooding with probabilistic pruning, anycast, multi-hop routing, application level multicast, tunnelling, clustering, delay-tolerant routing, and distributed hash table protocols. Some of the CF's generic APIs are as follows: `send(message, address)` is used for unicast; `sendToAll(message)` for broadcast; `sendToGroup(message, topic)` for multicast; `sendByKey(message, key)` for DHT; and `send(message)` for content-based routing.

Interaction Services. This CF supports an extensible set application-level "interaction paradigms" that may be layered on top of the Network Services CF. Examples of plug-ins accepted by this CF include: tuple spaces, reliable multicast, publish-subscribe and event notification, remote procedure call, etc. Many such plug-ins can coexist depending on application needs. Furthermore, one plug-in may exploit another (e.g. RPC may exploit a tuple-space component which in turn may be built on top of publish-subscribe).

Location Services. This CF is responsible for elaborating the physical or logical position of the host it is running on. It may exploit data acquired by a GPS sensor if present. Otherwise it may adopt other plug-in strategies; e.g. by interacting with GPS-equipped nodes.

Advertising and Discovery Services. Analogously to the way in which the runtime's registry allows applications to reason about the components available *locally*, the advertising and discovery CF allows applications to reason about *remote* components.

Components that wish to advertise their presence in the environment are called *Advertisable* components. Examples include codec repositories and general services. An Advertisable component implements the *Advertisable* interface, which enables the component to export a message that can be used for advertising. Plug-in advertising "strategies" (e.g. based on uPnP, SLP etc) are then represented as *Advertiser* components. These are responsible for taking the messages of advertisable components and, after potentially transforming them into other formats, using these as advertisements. There can be any number of Advertiser components installed in the CF.

Similarly, discovery "strategies", which allow clients to reason about the advertisable components that have been found remotely, are encapsulated as *Discovery* components.

Coordination Services. This CF hosts a range of plug-in protocols that are used to coordinate in various ways between RUNES nodes—especially between sensor network nodes. Examples are as follows:

Wake-up Coordination and Clustering Sensor motes typically employ a range of strategies to reduce bandwidth and save energy. Wake-up coordination strategies try to prevent redundant transmissions by ensure that transmitting and receiving nodes are both powered up at the same time. Similarly, a range of clustering strategies can be employed for data aggregation and energy-efficient communication—e.g. based on geographic proximity or other criteria such as battery level and sensor types. All such policies employ distributed coordination and can thus be provided as plug-ins under this heading.

Distributed Task Scheduler These plug-ins are responsible for allocating tasks on different nodes according to their specific capabilities and energy statuses. Inputs from the application include application structure, data/control flow, resource requirements, and constraints on latency, reliability and energy use. Distributed task schedulers are also responsible for managing the trade-off between energy consumption and application performance.

B. Examples of RUNES Meta-models

We now give a number of examples of reflective meta-models. Note that all of these meta-models can be loaded/unloaded on demand and thus only consume resources when and where actually needed.

First, here are three examples of generic meta-models that are scoped within a single capsule: The *architecture meta-model* exposes the compositional topology of a system of deployed components in a capsule as a casually-

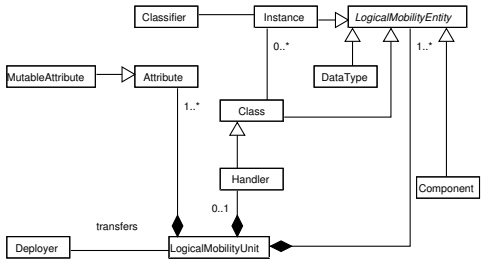

Fig. 2. A RUNES Reconfiguration Meta-Model.

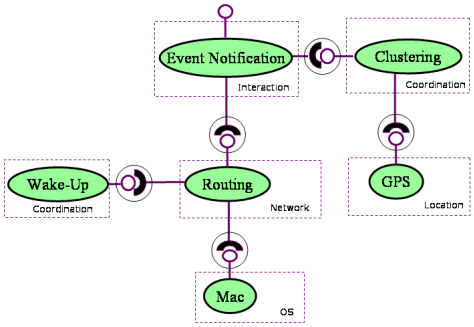

Fig. 3. Publish/Subscribe

connected graph structure. It also exposes the nested structure of composite components and CFs. It enables one to inspect, adapt and extend component topologies through high-level operations on nodes (components) and arcs (bindings). The *interception metamodel* allows one to interpose interceptors at bindings between component interfaces for purposes such as monitoring, debugging or security. The *interface metamodel* allows one to discover information about interface types at runtime and to invoke interface instances that are dynamically discovered at runtime. Note that these meta-models rely on special support from the runtime API (e.g. the architecture meta-model needs to be informed every time a component is created or destroyed); we have glossed over these details in this short paper.

As a more detailed example, we now describe a meta-model that can be used to manage system reconfiguration in a *distributed* environment. A high level view of this reconfiguration meta-model, which is based on [4], is given in Figure 2. The meta-model is based on principles of *code mobility* [5] and can be used to dynamically transfer code, state and data between RUNES nodes. The meta-model is *symmetric*, meaning that it allows nodes to act both as senders and receivers. The meta-model defines a *Logical Mobility Entity* (LME), as a generalisation of code, state, components or data. *Code* refers to, for example, classes or scripts; *state* refers to the state of executing code at a particular instance in time (for example, the value of variables); *components* refer to the collection of code elements and metadata that make up a component; and *data* refers to resources that other LMEs may reference (e.g. application profiles, files, etc.)

The meta-model also defines a *Logical Mobility Unit* (LMU) which is a collection of an arbitrary number of LMEs, and is the basic unit of deployment. As such, an LMU can contain anything from an individual class, to a collection of components. LMUs are described using a collection of $<key, value>$ attributes similar to those associated with the RUNES component model entities, and are transfered and deployed by so-called *Deployers*. These are responsible for sending and receiving requests, and for packing, serialising, transferring and deploying LMUs. LMUs do not necessarily have to be requested by their intended recipients; they can be imposed on the recipient by a controlling third party. LMUs comprehend both *physical* and *logical* destinations. The physical destination is the address of the recipient node. Valid logical destinations of an LMU are CFs, which allows for deploying components in particular CFs, and individual components, which allows for dynamically updating a single component.

An LMU can optionally encapsulate a *Handler* which represents code that can be used by the recipient node to deploy the contents of the LMU in a customised manner.

C. Example: Event Notification

To show how CFs can work together we now describe how an Event Notification service may be implemented, deployed and adapted in our system (see Figure 3).

The key component is an *Event Notification* component that is encapsulated within the Interaction Services CF. This exposes application interfaces to publish events and to subscribe to classes of relevant events. It also has receptacles for an appropriate network service (provided by the Network Services CF) to enable network level communication.

On deployment, the Event Notification component uses the architecture and interface meta-models to discover which network services are currently available, and to establish an optimal strategy accordingly. For example, it might bind to a content-based routing component that happens to be present, which will directly support the required content-based routing. Otherwise, it might have to directly deal with events and subscriptions itself, and rely for network communication on a more primitive network service such as generic multi-hop routing. Or, alternatively, if a flooding-based network service is available, events can be delivered to all nodes in the network and the Event Notification component itself can be responsible for performing the matching against its own subscriptions.

The architecture and interface meta-models might also be used to check whether a clustering component is available in the Coordination Services CF and, if so, to investigate which interfaces it provides (e.g. it may or may not offer location-based aggregation, depending on whether the host is equipped with a GPS-sensor). Given this information, the Event Notification component may refine its application interfaces by enabling the application to specify a subscription scope or a desired form of event aggregation.

Finally, in the operational phase, dynamic aspects of the

architecture come into play. For example, the interception meta-model might be used to detect a change in context from an ad-hoc networking environment to an infrastructure-based environment. On this basis, the Event Notification component might decide to use the architecture or reconfiguration meta-model to switch to a new and more appropriate network service.

IV. Related Work

There is a substantial body of literature on reconfigurable middleware systems. For example, *Gravity* [6] is a component model built on top of the Open Services Gateway Initiative (OSGi) Framework [7] (OSGi is a commercial framework for the Java platform which allows providers to deliver services to consumer devices attached to a residential network and to manage those devices remotely). *P2PComp* [8] is a lightweight service-oriented component model for mobile devices, which is also built using OSGi; it provides location independent synchronous and asynchronous communication between components. The *Dynamically Programmable and Reconfigurable Software* (DPRS) architecture [9] is a component-based design for dynamically programmable and reconfigurable systems. *PCOM* [10] is a distributed component model for pervasive computing. It allows for designing applications as a collection of potentially distributed components, which make their dependencies explicit. If those dependencies are invalidated, PCOM can attempt to automatically adapt by detecting alternatives according to various strategies. *FarGo-DA* [11] is a distributed component model that uses logical mobility to allow disconnected operation. The *Software Dock* [12] is an agent-based software deployment network that allows negotiation between software producers and consumers. *THINK* [13] presents an approach for building component-based operating system kernels. And finally, *one.world* [14] is a system for pervasive applications that supports dynamic service composition, migration of applications and discovery of context.

The main difference between RUNES and the approaches outlined above is *generality*: RUNES is a generic software fabric that is designed from the ground up to be implementable on a wide range of devices, and the primitives we provide are not limited to, for example, disconnected operation or software deployment, but can indeed be used to build such services.

V. Conclusions

In this position paper we have described the RUNES approach to the development of software for networked embedded systems. This employs a uniform "software component" abstraction which can be variously realised in various types of devices. Then, on top of the basic component model, we layer the notions of reflective meta-models and component frameworks. We have shown in the paper how these abstract notions can be instantiated as useful and general functionality, both local and distributed, in adaptive networked embedded environments.

In our future work, we want to investigate the practicality of our approach in a range of environments. In particular, we want to implement the runtime API on top of the Contiki OS [15] on sensor motes and thereby validate that the approach is viable even in extremely resource-constrained environments. Then we want to use the approach to develop application level functionality in terms of a emergency fire-fighting scenario that we have developed which employs at least three types of device: sensor motes, mobile PDAs and central control computers on the fixed network.

Acknowledgements: The authors would like to thank their partners in the RUNES Project and to acknowledge the financial support given to this research by the European Commission.

References

[1] G. Coulson, G. S. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama, "A Component Model for Building Systems Software," in *Proceedings of IASTED Software Engineering and Applications (SEA'04)*, Cambridge MA, USA, November 2004.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[3] P. Maes, "Concepts and experiments in computational reflection," in *Proc. of the OOPSLA-87: Conference on Object-Oriented Programming Systems*, Languages and Applications, Orlando, FL, 1987, pp. 147–155.

[4] 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: Springer, October 2004.

[5] A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, May 1998.

[6] H. Cervantes and R. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," in *Proceedings of the 26th International Conference of Software Engineering (ICSE 2004)*. Edinburgh, Scotland: ACM Press, May 2004, pp. 614–623.

[7] The OSGi Alliance, "The OSGi framework," http://www.osgi.org, 1999.

[8] A. Ferscha, M. Hechinger, R. Mayrhofer, and R.Oberhauser, "A Light-Weight Component Model for Peer-to-Peer Applications," in *2nd International Workshop on Mobile Distributed Computing*. IEEE Computer Society Press, March 2004.

[9] M. Roman and N. Islam, "Dynamically Programmable and Reconfigurable Middleware Services," in *Proceedings of Middleware '04*, Toronto, October 2004.

[10] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - A Component System for Pervasive Computing," in *Proceedings of the 2nd International Conference on Pervasive Computing and Communications*, Orlando, Florida, March 2004.

[11] Y. Weinsberg and I. Ben-Shaul, "A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices," in *Proceedings of the 24th International Conference on Software Engineering*, May 2002, pp. 374–384.

[12] R. S. Hall, D. Heimbigner, and A. L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," in *Proceedings of the 1999 International Conference on Software Engineering*. IEEE Computer Society Press / ACM Press, 1999, pp. 174–183.

[13] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, "THINK: a software framework for component-based operating system kernels," in *2002 USENIX Annual Technical Conference*. Monterey, CA: USENIX, June 2002, pp. 73–86.

[14] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall, "A system architecture for pervasive computing," in *Proceedings of the 9th workshop on ACM SIGOPS European workshop*. ACM Press, 2000, pp. 177–182.

[15] A. Dunkels, B. Groenvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, Nov. 2004.