The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario

Paolo Costa[*], Geoff Coulson[#], Richard Gold[†], Manish Lad[†], Cecilia Mascolo[†], Luca Mottola[*], Gian Pietro Picco[*‡], Thirunavukkarasu Sivaharan[#], Nirmal Weerasinghe[#], and Stefanos Zachariadis[†]

[#]Dept. of Computing
Lancaster University, UK
{geoff,n.weerasinghe,t.sivaharan}
@comp.lancs.ac.uk

[†]Dept. of Computer Science
University College London, UK
{r.gold,m.lad,c.mascolo,s.zachariadis}
@cs.ucl.ac.uk

[*]Dept. of Electronics and Information
Politecnico di Milano, Italy
{costa,mottola,picco}@elet.polimi.it

[‡]Dept. of Information and Communication Technology
University of Trento, Italy
picco@dit.unitn.it

Abstract

Due to the inherent nature of their heterogeneity, resource scarcity *and* dynamism*, the provision of middleware for future networked embedded environments is a challenging task. In this paper we present a middleware approach that addresses these key challenges; we also discuss its application in a realistic networked embedded environment. Our application scenario involves fire management in a road tunnel that is instrumented with networked sensor and actuator devices. These devices are able to reconfigure their behaviour and their information dissemination strategies as they become damaged under emergency conditions, and firefighters are able to coordinate their operations and manage sensors and actuators through dynamic reprogramming. Our supporting middleware is based on a two-level architecture: the foundation is a language-independent, component-based programming model that is sufficiently minimal to run on any of the devices typically found in networked embedded environments. Above this is a layer of software components that offer the necessary middleware functionality. Rather than providing a monolithic middleware 'layer', we separate orthogonal areas of middleware functionality into self-contained components that can be selectively and individually deployed according to current resource constraints and application needs. Crucially, the set of such components can be updated at runtime to provide the basis of a highly dynamic and reconfigurable system.*

1 Introduction

Future networked embedded infrastructures will have to support very challenging application scenarios. One such scenario, quite representative of the field and constituting the applicative focus of our RUNES (Reconfigurable Ubiquitous Network Embedded Systems) project[1], involves a road tunnel that is instrumented with networked embedded sensors, actuators, and larger, more powerful, devices. The latter act as gateways and allow the sensors to report monitored readings both directly to the actuator systems and to a tunnel control centre. The system allows to detect and react to emergency situations such as fire or chemical spillage. In an emergency, firefighters enter the tunnel in groups. As the situation unfolds, the embedded sensor/actuator network reconfigures itself as devices fail, and devices carried by the firefighters spontaneously inter-work with each other and with the embedded devices to provide the firefighters with appropriate information and command/control capability.

Scenarios such as this are essentially characterised by heterogeneity, resource scarcity and dynamism. In terms of *heterogeneity*, the devices employed range from tiny sensors to controller PCs and the PDA-class devices carried by the firefighters. These different devices employ a variety of power sources, run different operating systems, and are programmed in different languages. Furthermore, they interact using a range of network types including both wired and wireless networks running in both infrastructure and ad-hoc modes, and a range of higher-level interaction paradigms such as messaging, RPC, and publish-subscribe. *Resource scarcity* is clearly an issue for many of the device classes involved. Apart from the obvious issues of power and CPU speed, memory can be a significant limitation that can severely constrain the 'intelligence' of devices and also limit their capability to buffer messages. Finally, such scenarios are inherently *dynamic* due to changing envi-

[1]http://www.ist-runes.org

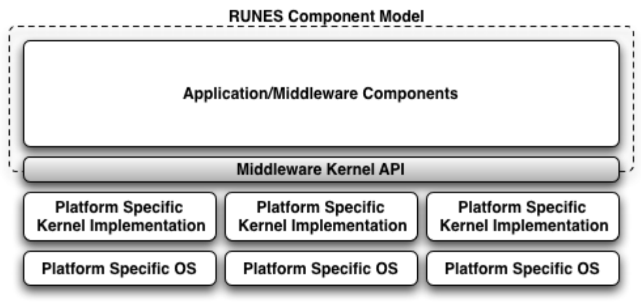
RUNES Component Model
Application/Middleware Components
Middleware Kernel API
Platform Specific Kernel Implementation
Platform Specific Kernel Implementation
Platform Specific Kernel Implementation
Platform Specific OS
Platform Specific OS
Platform Specific OS

Figure 1. The RUNES software architecture.

ronmental conditions. Obvious examples from our disaster management scenario are loss of devices due to fire damage; loss of network connectivity; and the need to spontaneously create new patterns of connectivity as firefighters move around the tunnel. Such situations require the system to be capable of dynamically reconfiguring itself along several different dimensions such as reconfiguring the network topology, loading new functionality onto devices, and offloading functionality as resources dwindle.

In this paper we focus on the provision of *middleware* for such scenarios. We first observe that traditional solutions such as CORBA, Jini, .NET etc. are ill-matched because they lack sufficient support for heterogeneity, resource scarcity and dynamism. We therefore take a "clean-slate" approach in the shape of the architecture shown in Figure 1. The foundation for the architecture is provided by a component-based programming model, provided to the programmer through a *middleware kernel API*. Since the model effectively captures the essence of the required functionality in only a handful of concepts, the kernel supporting it is simple enough to be easily implementable on any of the devices typically found in our target scenarios. This API is then employed to build, at the upper level, a composition of middleware and application-level *software components* that offer the necessary middleware and application functionality. Rather than providing a monolithic middleware "layer", we factor orthogonal areas of middleware functionality into self-contained components that can be selectively and individually deployed and composed according to current resource constraints and application needs. For example, some devices might require only a basic communication component that provides unreliable messaging, whereas others might require a more sophisticated publish-subscribe service that can be realised by composing additional components on top of the base one. Crucially, the set of such components can be updated at runtime to provide the basis of a highly dynamic and reconfigurable system.

The contributions of this paper are as follows:

- a middleware approach that addresses the above-mentioned concerns of heterogeneity, resource scarcity and dynamism;
- a concrete implementation of the approach in three representative deployment environments, viz. Java, C/Unix, and severely resource-constrained sensor devices running the Contiki [9] operating system;
- an implementation of the disaster management application scenario described earlier, which provides a concrete way to assess the effectiveness of the approach;
- an experimental, quantitative evaluation of our three specific implementations showing their ability to accommodate heterogeneity, resource scarcity and dynamism with reasonable performance.

The remainder of the paper is structured as follows. First, we expand in Section 2 on our motivating disaster management scenario. In Section 3, we outline our component model, its API, and its deployment in the three different environments mentioned above. We then, in Section 4, describe the design of a specific middleware solution for the disaster management scenario, and in Section 5 we present our empirical evaluation. Finally, we compare our approach to related work in Section 6, and offer our conclusions in Section 7.

2 The Road Tunnel Application Scenario

Over 200 people have died in Europe in road tunnel fires during the last decade. There is therefore considerable interest in applying technology to improving safety in road tunnels. However, as reported in an article on a Berlin tunnel in Risks Digest getting the technology right is very difficult and the current state of the art is not very advanced [1].

In our futuristic scenario, we envisage a road tunnel that is equipped with networked embedded devices that monitor environmental conditions such as temperature, humidity and air quality, and actuate tunnel safety systems such as sprinklers, ventilators and road signage. The system also incorporates larger, more powerful, devices which act as gateways and allow the sensors to report monitored readings both directly to the actuator systems and to a tunnel control centre.

When an accident occurs, the system's first responsibility is to detect and report the accident and carry out any automated emergency sequences. In addition, some sensors, actuators or gateways may be damaged and the system must reconfigure itself to compensate for this. Eventually, a team of firefighters arrives. We envision the firefighters equipped with PDA-class networked devices capable of interacting directly with the tunnel system and also carrying

```
interface Capsule : {
  ComponentType load(in Pattern p);
  void unload(in ComponentType t);
  Component instantiate(in ComponentType t);
  void destroy(in Component c);
  Connector connect(in Interface i, in Receptacle r,
                    in ConnectorFactory cf);
  void setAttribute(in Entity e, in Attribute a);
  sequence<Attribute> getAttributes(in Entity e,
                                    in Pattern p);
  sequence<Entity> getEntities(in Pattern p);
};
```

Figure 2. The Kernel API.

body sensors to report vital signs to other workers to ensure that they are rescued when needed. At this point, the tunnel system plays the role of a tool that can be directly manipulated by the firefighters. For example, it can be selectively queried by the firefighters to help them operate in the poor visibility conditions, and the firefighters can directly control the actuator devices.

In the next two sections we first expand on the fundamentals of our middleware approach and then show how the approach can be used to build an application for the road tunnel scenario.

3 The RUNES Middleware Foundations

In this section, we first describe our software component model and its associated API. As shown in Figure 1, this API is provided at runtime by the middleware kernel. We then discuss the middleware kernel implementation for three very different platforms and briefly comment on components we have developed.

3.1 The Middleware Kernel

Our component model[2] comprises the following elements: *components*, *component types*, *interfaces*, *receptacles*, *connectors*, *connector factories*, *attributes* and *capsules*. The API associated with the model is defined in Figure 2 in terms of the OMG's Interface Definition Language (IDL). In addition, the relationships between the various elements is shown diagrammatically (using UML) in Figure 3.

In the model, *components* are the basic runtime units of encapsulation and deployment. They are instantiated at runtime from *component types*, such that each component type can be used to create multiple component instances at runtime. This is performed using the `instantiate()` operation in Figure 2. Components can be deleted as well, using `destroy()`. Component types can themselves be dynamically loaded and unloaded at runtime (see `load()` and `unload()`), which provides the basis for the dynamic nature of our programming model[3].

Components offer their functionality through one or more *interfaces* each of which is defined in a programming language independent manner as a set of types and operation signatures. In addition, components that have dependencies on other components can express these dependencies in terms of one or more *receptacles*. This capability is of considerable help when components are dynamically deployed, as the required deployment environment of the new component is made clear and explicit. Such a component must have each of its receptacles connected (using `connect()`) to a corresponding interface on some external component before it can execute. This connection between a receptacle and an interface is explicitly represented in the model through a so-called *connector*, which is itself a component and therefore can be deleted using `destroy()`.

The model also incorporates the notion of *connector factories*. These are components that create connectors that embody a specific piece of behaviour to be invoked every time a call is made over a given receptacle/interface connection. In this way, connectors may encapsulate arbitrary functionality and can thus be used to perform such functions as monitoring or intercepting communications between their associated receptacle and interface. Connector factories are passed as arguments to `connect()`; or, if a null argument is passed, a "default" connector factory is used which binds the receptacle directly to the interface. Note that connector factories are *not* normally used to abstract over network communications; rather, they are intended for "local" use only. Network communication is assumed to be encapsulated within middleware components (see Section 4) and is thus transparent to the component model itself.

All of the foregoing entities (i.e., components, component types, interfaces, receptacles, and connectors) may be annotated with *attributes*. These are key/value pairs that can be used to express arbitrary meta-data. Attributes are managed using `setAttribute()` and `getAttributes()`. Finally, all of the foregoing entities reside inside a *capsule* which serves as a runtime component container, providing name space functionality. A capsule is typically implemented as an operating system address space although this is not mandatory. All the entities currently inside a capsule can be enumerated using the `getEntities()` operation.

It is notable that the component model can be managed using only *eight* operations as illustrated in Figure 2. This

[2]A preliminary version of the model appeared in [6]. The one presented here is richer and is furthermore complemented by details on kernel implementations, on the application scenario, and on evaluation.

[3]The 'pattern' argument to `load()` is simply a flexible way of specifying a component type.

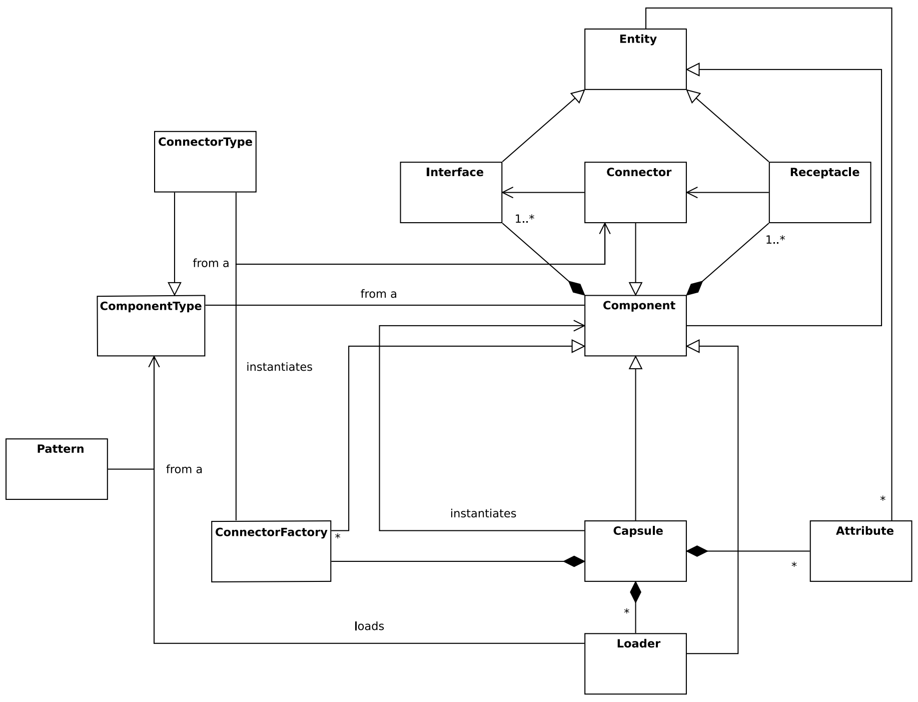

Figure 3. The RUNES component model.

enables an easy and lightweight implementation of the kernel mechanisms on a wide range of platforms, as illustrated next.

3.2 Kernel Implementations

We now describe concisely how the core abstractions defined by our component model are realised in three different implementations: *(i)* a Java-virtual-machine-based implementation; *(ii)* a C/Unix-based implementation; and *(iii)* an implementation based on tiny embedded devices running the Contiki [9] operating system.

Component Types and Components. In the Java implementation, component types are straightforwardly represented as classes that inherit from a specific abstract class. This approach allows us to "factor out" the code needed to support component instantiation and destruction. Therefore, components can be realised simply as objects instantiated from a class representing a component type, and the `load()` operation is simply implemented using the default Java class loader. In the C/Unix implementation component types are represented as Unix "shared objects" compiled from source files conforming to a specified structure. The `load()` operation is implemented in terms of the native load/link facilities provided by the operating system, e.g., using `dlopen()`, and instantiation amounts to allocating a struct containing per-component state. Each interface operation defined in a component type (realised as a C function) takes as its first argument a pointer to this per-component struct so that the particular component instance being invoked can be determined. In the Contiki implementation, component types are similarly implemented as C source files which map to Contiki "services"; and the Contiki dynamic loading facility is used. Because Contiki supports only a single instance of a given type of "service", the `instantiate()` operation currently only returns a newly instantiated component *once* for each component type. We are currently looking into removing this limitation.

Interfaces, Receptacles and Connectors. In the Java environment, interfaces are trivially implemented as Java interfaces, whereas receptacles are implemented as Java objects. Component types contain initialisation code to create the appropriate receptacles at component instantiation time. In the C/Unix environment both interfaces and receptacles are represented as C structs. Both contain an array of function pointers. In the case of an interface, these pointers point at the target operations (C functions). In the case of a receptacle, they are assigned during `connect()` either directly to the function pointer values in the associated interface, or indirectly via functions within the specified connector that contains some intermediate functionality. In the Contiki environment, a similar approach is adopted. In the Java and C/Unix environments we provide the 'full' semantics

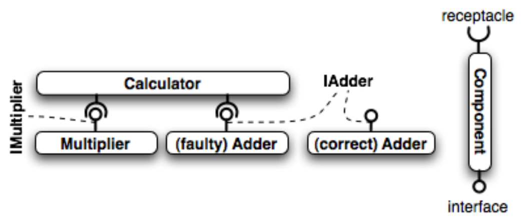

Figure 4. A component-based calculator.

```
import runes.kernel.Interface;
public interface IMultiplier extends Interface {
  public int multiply(int x, int y); }
```

Figure 5. Java kernel: interface for a generic multiplier component. `Interface` is a tagging interface every RUNES interface must extend.

of connectors, i.e., we provide the ability to employ user-defined connector factories to customize their behaviors as described above. Currently we do not provide this functionality in the Contiki environment, but there is no a priori reason why the Contiki implementation could not be extended in this way.

3.3 Using Components: An Example

We have developed a range of application and middleware components on top of the kernel API discussed above. Some of these are discussed in our previous paper [6]. Here we describe the implementation of a simple application to illustrate the use of the abstractions and API described above.

Our example application consists in a component-based calculator, designed according to the component configuration shown in Figure 4. The *Calculator* component offers an interface providing operations to multiply or add two integers. Each of these operations is implemented by a dedicated component, connected to the *Calculator* through a dedicated connector. At some point, the system recognizes the component initially employed to implement the add operation as faulty. Therefore, the system dynamically replaces it with a different *Adder* component implementing the same interface.

Developers start writing the interfaces expressing the operations requested/exported by components. For instance, the *Multiplier* component relies on a single interface containing an operation to multiply two integers, as illustrated in Figure 5 using Java. The operation signatures must then be implemented within the actual components, possibly along with initialization and destruction routines executed at component instantiation and destruction time, respectively. This is illustrated in Figure 6 in the case of the *Multiplier* component.

Components are wired together so that the *Calculator* exploits the other components to implement its operations. This wiring is performed using the primitives provided by the *Kernel* API, as shown in Figure 7. In our example, this involves creating various component instances, and connecting the *Calculator* component to a pair of components implementing the *IMultiplier* and *IAdder* interfaces.

```
public class Multiplier extends BaseComponent
                        implements IMultiplier {
  public void construct() {
    System.out.println("Multiplier instantiated"); }
  public void destroy() {
    System.out.println("Multiplier destroyed");}
  public int multiply(int x, int y) {
    return x * y;}
}
```

Figure 6. Java kernel: implementation of a simple multiplier component. `BaseComponent` is an abstract class every Java component must inherit from.

```
// Loading and instantiating a Calculator component
ComponentType calcT =
    capsule.load("sampleApp.Calculator");
Component calc = capsule.instantiate(calcT);
// Loading and instantiating a (faulty) Adder component
ComponentType adderTFaulty =
    capsule.load("sampleApp.FaultyAdder");
Component adder = capsule.instantiate(adderTFaulty);
...
// Retrieve interfaces and receptacles as attributes
Interface adderIf = (Interface) capsule.
    getAttribute(adder,"INTERFACE-sampleApp.IAdder");
Receptacle calcAdderRecpt = (Receptacle) capsule.
    getAttribute(calc, "RECEPTACLE-sampleApp.IAdder");
...
// Connecting the Calculator to the (faulty) Adder
Connector calcToAdder =
    capsule.connect(adderIf, calcAdderRecpt);
```

Figure 7. Java kernel: instantiating and connecting components for the calculator application.

```
// Loading and instantiating a new Adder component
ComponentType adderTCorrect =
    capsule.load("sampleApp.CorrectAdder");
Component adder = capsule.instantiate(adderTCorrect);
// Retrieving the new Adder interface
Interface adderIf = (Interface) capsule.
    getAttribute(adder,"INTERFACE-sampleApp.IAdder");
// Destroying the Calculator-FaultyAdder connector
capsule.destroy(calcToAdder);
// Connecting the Calculator to the correct Adder
calcToAdder = capsule.connect(adderIf, calcAdderRecpt);
```

Figure 8. Java kernel: replacing the adder component in the calculator application.

The *Kernel* API is also used to replace the faulty adder component with a correct one. This is done by first destroying the *Connector* binding the *Calculator* and the *FaultyAdder*, and then reconnecting the former to a newly instantiated *Adder* component, as exemplified in Figure 8.

In its simplicity, the above example shows the power and ease of use of the kernel API. The next section illustrates how we leveraged off these characteristics to address the challenges of our reference scenario.

4 The RUNES Middleware in Action

Using our three middleware kernel implementations, we have developed a set of middleware and application components that collectively address the road tunnel disaster management scenario outlined earlier. The overall design of the resulting application is depicted in Figure 9.

The application is structured as follows. TMote Sky [23] nodes running the Contiki-based kernel support a *Data Acquisition* component and a *Data Dissemination* component that together monitor and disseminate environmental conditions in the tunnel. These report, via gateways running the C/Unix kernel and supporting a *Packet Forwarding* component, to a central control station that includes a *Data Logging* component running on a PC that runs the Java kernel. The communication is handled by an underlying *μAODV* component which provides multi-hop routing.

When an emergency occurs, the Data Acquisition components respond initially by sending readings more frequently. In addition, the μAODV component has the ability to automatically recover from damage to either sensors or communication paths. Eventually, firefighters arrive equipped with mobile, wireless devices, forming a mobile ad-hoc network. The firefighters' devices instruct the sensors to send their readings direct to the firefighter as well as to the Data Logger and also a *Publish-Subscribe* [10] component that helps the firefighters coordinate their actions. The firefighters additionally run a *Deployment* component that has the capability to dynamically deploy a Contiki version of the Publish-Subscribe component directly onto the sensor devices so that the latter can start broadcasting directly to any nearby firefighters who subscribe to relevant events, e.g., temperature readings above a safety threshold. The Deployment component first checks if the sensors within range already run the Publish-Subscribe component. If not, the owning firefighter is prompted about the possibility of uploading the component on those sensor devices still lacking it. If there is no space on a sensor for the Publish-Subscribe component, the original Data Dissemination component can be removed. All of this behaviour is under the control of the firefighters who interact with their devices using a GUI component. Table 1 summarises the configuration of the devices involved as the situation unfolds.

	Device	**Kernel Platform**	**Middleware Components**
Step 1 Quiescent conditions	Sensor	Contiki	Data Acquisition Data Dissemination μAODV
	Gateway	C/Unix	Packet Forwarding
	Control center	C/Unix	Data Logging
Step 2 Fire detected	Firefighter	Java	Publish-Subscribe GUI Component
Step 3 Firefighters reconfigure sensors	Sensor	Contiki	Publish-Subscribe
	Firefighter Firefighter	Java	Publish-Subscribe Deployment

Table 1. Configuration of the application as the scenario unfolds.

The Publish-Subscribe component is the most complex of the components described above and deserves further explanation. The component employs a layered architecture, in which two sub-components take care respectively of the two concerns relevant to dealing with host mobility, i.e., overlay maintenance and route reconfiguration on top of the overlay. The separation of these two concerns is especially beneficial in allowing independent customisation of these two aspects. In more detail, the first sub-component takes care of creating and maintaining a tree-shaped overlay based on the algorithm described in [24]. The second sub-component is then in charge, using the mechanism described in [25], of setting up message routes on top of the overlay, and reconfiguring these routes in case of topology change.

Note that the application provides a clear illustration of the benefits of our middleware approach. First, a unified component-based software development approach is adopted regardless of the type of device involved. Second, the component approach encourages the development of independent pieces of functionality that can be composed in various useful ways depending on context. Third, the dynamic loading capability relaxes the need to anticipate all the functionality that will be needed on a node. This is especially beneficial for resource-constrained devices on which it may not be possible to fit all the components required at any one time. Fourth, the dynamic (re)connection capability makes it possible for newly deployed components to interact in complex ways with the existing components in a type-safe manner. For example, initially, the Data Acquisition component is bound to the Data Dissemination component; however, when the Publish-Subscribe component is uploaded, it is dynamically rebound to the latter.

5 System Evaluation

This section assesses the effectiveness of our middleware in coping with heterogeneity, resource scarcity and

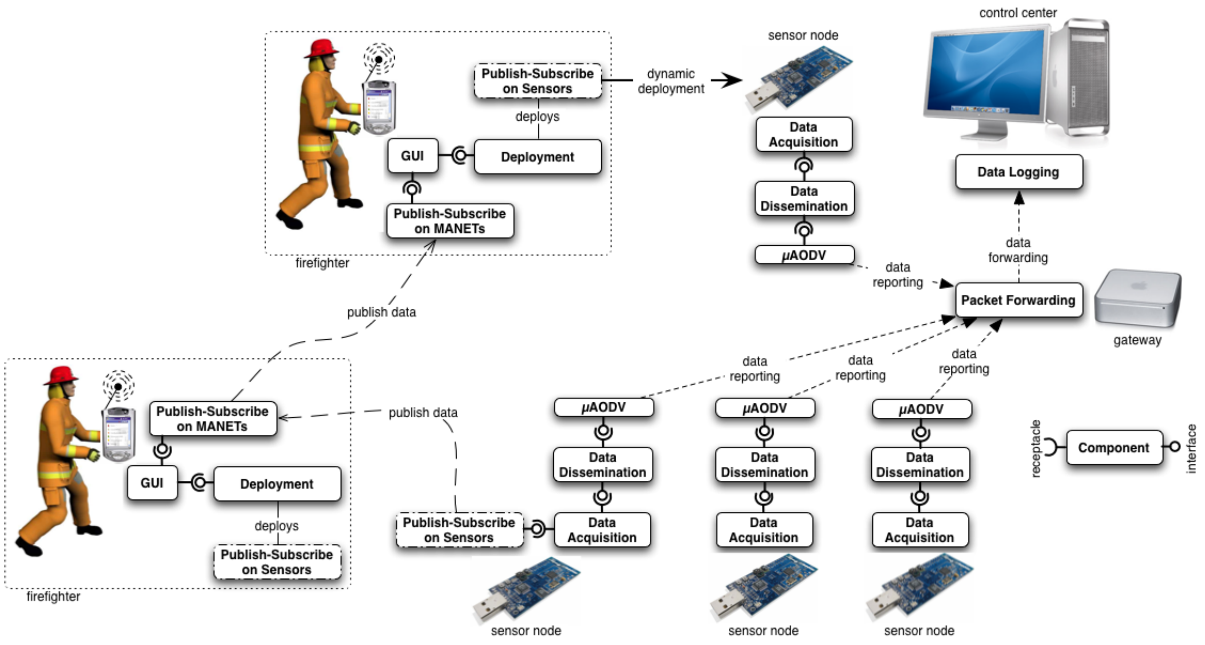

Figure 9. Fire in a road tunnel: application design.

dynamism, and also assesses its specific competence for our reference disaster management scenario.

5.1 Middleware Kernel Evaluation

First we present an evaluation of our three middleware kernel implementations. For the Java implementation we used Sun JVM v5.01a running on Linux Gentoo 2006.1 on a Pentium 4 3.2Ghz with 1 GB RAM. For the C/Unix implementation we used and 2.4 Ghz P4 running Linux 2.6.12-9-386; and for the C/Contiki implementation we used TMote Sky motes [5] with a 250 kbps radio, 10 KB RAM, 48 KB flash, and 1 MB storage.

Metrics. To demonstrate the ability of the middleware to support heterogeneity and resource scarcity, we measure the *kernel memory footprint*, i.e., the data and code memory footprint consumed by the run-time support for our component model. In addition, to evaluate the memory overhead required to represent the component, interface and receptacle concepts from the programming model, we measure the *memory footprint of a null component* and the *per-interface, per-receptacle memory footprint*. A null component is a component with no interfaces/receptacles and null initialisation/destruction routines.

To investigate the dynamic aspects of the middleware, we consider the overhead of *null operation calls through a default connector*. A null operation is one with no in-/out parameters performed across a connector without intervention in the control flow, and introduces some overhead w.r.t. "native" operation calls (e.g., a method invocation in Java). This measure represents the run-time overhead of introducing connectors in the programming framework. We also consider the operations needed to dynamically modify the software running on a node. To that end, the kernel must *load* a new component, *instantiate* it, and *connect* the new instance to an existing component. In the case of the Java and C/Unix kernels, we measured each of these aspects separately, using a null component. We note that the fine-grained time aspects cannot be measured on the motes due to timer service limitations.

Results and Discussion. Our approach addresses heterogeneity effectively. This is shown by implementing the same software component model on a variety of devices, ranging from powerful desktop PCs to resource-constrained devices. Different programming languages and concurrency models have been used on different platforms. Our support of heterogeneity is further demonstrated by the relative sizes of the different middleware kernel implementations, shown in Table 2. This highlights that our implementations scale down to severely constrained devices.

Even on the most resource-constrained of our platforms, the TMote Sky motes, the kernel footprint of 780 bytes is less than 1% of the total available flash memory of the motes (48KB internal and 1MB external flash memory). The overhead due to the introduction of components, interfaces and receptacles in the programming model of Contiki is negli-

Performance Measure (Memory Footprint)	**Java**	**C/Unix**	**C/Contiki**
Kernel Code	14.65 KB	16 KB	780 bytes
Kernel Data	840 bytes	4 KB	52 bytes
Null Component Data	544 bytes	24 bytes	9 bytes
Per-Interface Data	200 bytes	40 bytes	2 bytes
Per-Receptacle Data	264 bytes	22 bytes	2 bytes

Table 2. Memory overhead.

Performance Measure	**Java**	**C/Unix**	**C/Contiki**
Overhead of null Calls (DefaultConnector)	158.93%	99.84%	137.5%
Component Loading Time	0.0006 ms	0.2116 ms	2.4973 s
Component Instantiation Time	0.0047 ms	0.7674 ms	N/A

Table 3. Run-time overhead

gible with respect to the amount of RAM (10KB) available on the TMote Sky motes onto which they would be loaded. This minimal overhead obtained is due to the simplicity of our component model. This enables software reconfiguration through simple, yet powerful, abstractions, that are easily implementable.

Table 3 reports on the dynamic aspects of our implementations[4]. The overhead introduced by null operation calls through default connectors may appear to be non-negligible with respect to their "native" equivalents. However, further investigation revealed that invoking a void Java method through a default connector takes only 23.5 μs, on average. Therefore, the time needed to execute an actual fragment of code inside the method body would consume the majority of the overall computation time, making the overhead of the connector negligible. Similar considerations apply for the C/Unix and the Contiki implementations.

The remaining data in the same table refers to the operations needed to change the software running on a node. Among these operations, component loading and instantiation are the most expensive, because of the work involved in transferring the component and creating data structures within the middleware kernel. Given the values obtained, and also considering that such operations should be triggered only when needed, we argue our kernel implementations are able to adapt sufficiently quickly to a changing environment.

5.2 Scenario-Based Evaluation

We now provide a basic evaluation of aspects of the road tunnel scenario reported in Section 4. These measurements were made on an experimental set-up consisting of a TMote Sky node representing a sensor device in the tunnel, and two laptops representing firefighter devices. More precisely, the firefighter devices each comprise a laptop plus a TMote Sky

[4]We executed 10,000 iterations and averaged the results.

Performance Measure	**Data Acquisition**	**Data Dissemination**	**Publish-Subscribe**
Source Lines of Code	287 lines	181 lines	197 lines
Memory Footprint	1462 bytes	738 bytes	772 bytes

Table 4. Application component size.

node attached to the laptop via a USB cable; the TMote Sky node simply forwards IP packets from the firefighter laptop to the tunnel sensor and vice versa. The sensor device runs the Contiki implementation of our middleware kernel, whereas the firefighter devices run the Java version.

First, we evaluated the sizes of some of the components running on the sensor device. The results in Table 4 show that these are negligible compared to the available resources of the TMote Sky motes. By adding together the footprints of the components and the middleware kernel, we see that the size of the sensor node installation is 3750 bytes. This is still less than 1% of the total memory available on a TMote Sky mote.

We also measured the lines-of-code and memory overhead for the Java Publish-Subscribe component on the firefighter devices. This amounts to 1327 lines of non-commented code, and occupies 8.23 KB of memory. Again, a very acceptable overhead.

Finally, we carried out some basic performance measures to confirm that the network overheads are sufficiently small for run-time reconfiguration to be feasible. To this end, we measured 2.07 seconds to deploy a null component onto the sensor device; 61.52 ms for an ICMP round-trip ping between the sensor device and a firefighter device; and 4.25 ms for a Publish-Subscribe message sent between firefighter devices. These figures indicate that the network overheads are indeed acceptable.

6 Related Work

There is a substantial body of literature on reconfigurable middleware systems. First, the RUNES middleware builds on our earlier work on the OpenCOM component model [7]. Compared to this earlier work, our middleware exhibits a richer and more coherent set of features, a sounder conceptual basis provided by the model we outlined in Section 3, and a more marked slant towards the requirements of networked embedded systems.

Other relevant component models exist. *Gravity* [4] is a component model built on top of the Open Services Gateway Initiative (OSGi) Framework [28]. *P2PComp* [12] 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 [26] is a component-based design for dynamically programmable and reconfigurable systems. *PCOM* [2] 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* [30] is a distributed component model that uses logical mobility to allow disconnected operation. The *Software Dock* [14] is an agent-based software deployment network that allows negotiation between software producers and consumers. *THINK* [11] presents an approach for building component-based operating system kernels. And finally, *one.world* [13] is a system for pervasive applications that supports dynamic service composition, migration of applications and discovery of context.

Other component based systems are targeted specifically at embedded systems. These include *Pebble* [21], *PECOS* [31], *PBO* [27], *SaveCCM* [15] and *Koala* [29]. Most of these are build-time only technologies—components are not visible at run-time and therefore these systems do not support dynamic reconfiguration, as we do. A further observation is that many of these embedded systems technologies (e.g., PBO, SaveCCM, and Koala) are tightly coupled to a specific underlying operating system and/or are programming language specific.

Existing middleware for WSNs (e.g., [16, 17, 20]) mostly addresses homogeneous systems and targets applications such as data collection and analysis. Therefore, it is unsuited to control applications like the ones we consider here, in which heterogeneous devices are deployed, and the system not only observes the environment, but also acts on it. Impala [19] is one system that does provide hooks for sensor actuation as a result of data sensing, but it targets homogeneous systems and does not provide a framework that can be used to develop components on multiple platforms. Similarly, most existing mechanisms for code deployment in WSNs (e.g., [18, 22]) cannot be used in our context, as they assume a single hardware platform, and are geared to distributing code to the whole system. Moreover, some of them do not even provide fine-grained control over the unit of deployment. For instance, [18] basically replaces the whole binary running on a node. Instead, our component model enables greater control over the unit of code distribution, ultimately achieving more efficient code deployment in terms of network load.

Finally, in terms of application scenarios, a number of middlewares targeting disaster, medical and emergency services have started to appear [8, 3]. However, we have not yet seen efforts offering a unique solution tackling the different requirements of heterogeneity, dynamism and resource variability.

In summary, there are two main differences between the approaches outlined above and our work. The first difference relates to *generality*: RUNES is a generic software fabric that is designed from the ground up to be implementable on a wide range of devices, and to allow the implementation of a large number of very different primitives. This is an essential requirement of pervasive applications such as disaster management. The second difference relates to our two-layer architecture in which systems are built by selecting (and dynamically reconfiguring) appropriate middleware and application components on top of the middleware kernel. This capability, lacking in other works, results in significantly greater flexibility than current systems offer.

7 Conclusions

In this paper we have described our approach to the provision of middleware for networked embedded environments and have demonstrated its application in a road tunnel disaster management scenario. We have shown in particular how our component model and its implementation provides a unified programming model over a wide range of devices including very small ones. Furthermore, the application of our approach in the highly heterogeneous road tunnel scenario has proved to be straightforward and to have low resource overhead.

As part of future work we will look into implementations of the middleware more tightly coupled with the operating system, into delivery of security properties and particularly access control, and into more sophisticated mechanisms for code deployment.

The middleware implementation is publicly available at `http://www.ist-runes.org/middleware`.

Acknowledgments: We would like to acknowledge the support of the EU through project IST RUNES. We also thank Björn Grönvall for his precious support for the Contiki implementation.

References

[1] ACM. The Risks Digest Forum on Risks to the Public in Computers and Related Systems: Computer closes Berlin tunnel again, July 2006.

[2] 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*, Florida, USA, 2004.

[3] D. Bottazzi, A. Corradi, and R. Montanari. Context-aware middleware solutions for anytime and anywhere emergency assistance to elderly people. *IEEE Communications Magazine*, 44(4):82 – 90, Apr. 2006.

[4] 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)*, pages 614–623, Edinburgh, Scotland, May 2004. ACM Press.

[5] M. Corporation. TMote Sky motes, 2006. `http://www.moteiv.com/`.

[6] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Network Embedded Systems. In *Proc. of* 16th *International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05)*. IEEE Press, Sept. 2005.

[7] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyam. A Component Model for Building Systems Software. In *Proc. IASTED Software Engineering and Applications (SEA'04)*, Nov 2004.

[8] O. Drugan, T. Plagemann, and E. Munthe-Kaas. Building resource aware middleware services over MANET for rescue and emergency applications. In *Proc. of* 16th *International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05)*. IEEE Press, Sept. 2005.

[9] A. Dunkels, B. Grönvall, 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*, Florida, USA, Nov. 2004.

[10] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2(35):114–131, June 2003.

[11] 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*, pages 73–86, Monterey, CA, June 2002. USENIX.

[12] 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 Press, March 2004.

[13] 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*, pages 177–182. ACM Press, 2000.

[14] 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*, pages 174–183. IEEE Computer Society Press / ACM Press, 1999.

[15] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren. SaveCCM – a component model for safety-critical real-time systems, Sept. 2004.

[16] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[17] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Networking*, 11(1):2–16, 2003.

[18] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proc. of the* 1st *Symp. on Network Systems Design and Implementation (NSDI04)*, 2004.

[19] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *Proc. of The ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, 2003.

[20] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[21] K. Magoutis, J. Brustoloni, E. Gabber, W. Ng, and A. Silberschatz. Building Appliances out of Reusable Components using Pebble. In *Proc. SIGOPS European Workshop*, pages 211–216, Kolding, Denmark, Sept. 2000. ACM Press.

[22] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O.Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proc. of the* 1st *European Workshop on Wireless Sensor Networks (EWSN06)*, 2006.

[23] MoteIV. Tmote sky specification. http://www.moteiv.com/products-tmotesky.php.

[24] L. Mottola, G.Cugola, and G. Picco. A Self-Repairing Tree Overlay Enabling Content-Based Routing in MANETs. Submitted for publication. Available at `www.elet.polimi.it/upload/mottola`, 2006.

[25] G. P. Picco, G. Cugola, and A. Murphy. Efficient content-based event dispatching in the presence of topological reconfigurations. In *Proc. of the* 23rd *Int. Conf. on Distributed Computing Systems (ICDCS03)*, pages 234–243, 2003.

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

[27] D. Stewart, R. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects. Technical Report CMU-RI-TR-93-11, Robotics Institute, Carnegie Mellon University, July 1993.

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

[29] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, Mar. 2000.

[30] 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*, pages 374–384, May 2002.

[31] M. Winter, T. Genbler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Muller, C. Stich, and B. Schonhage. Components for embedded software: the PECOS approach. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02)*, pages 19–26, Grenoble, France, 2002. ACM Press.