© 2006 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

# ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-Actuator Networks

J. Domaszewicz, M. Rój, A. Pruszkowski, M.Golański, and K. Kacperski
*Institute of Telecommunications, Warsaw University of Technology*
*ul. Nowowiejska 15/19, 00-665 Warsaw, Poland*
*e-mail: meag@tele.pw.edu.pl*

## Abstract

*The paper presents a programming model for a new pervasive computing middleware. The middleware, called ROVERS, targets an environment composed of tiny, resource-constrained, wirelessly communicating nodes embedded into everyday objects. The environment is heterogeneous in that each node is equipped with a unique set of sensors and actuators. The nodes establish an ad-hoc network and contribute their specific resources. The ROVERS layer transforms the network into a distributed pervasive computing platform. The ROVERS application is an evolving tree of cooperating, mobile micro-agents. The tree adapts to available resources and the current context. It is largely decoupled from the concept of the physical node. ROVERS provides the programmer with implicit resource discovery, inter-agent communications with logical addressing, minimization of application-generated traffic, ontology-driven representation of sensor and actuator resources, as well as support for component-based programming. The programming model lends itself to an implementation for a miniature operating system, like TinyOS.*

## 1. Introduction

This paper targets a pervasive computing environment composed of tiny, unobtrusive, resource-constrained nodes embedded into everyday objects. Each node contributes some resources to the computing environment. A node's resources can be classified into generic and non-generic.

Generic resources are those available at every node (e.g., computing capability, basic peripherals, and wireless connectivity). Non-generic resources are those available at some nodes, but not at others. A node's non-generic resources depend mainly on the functionality of the object the node is embedded in. Primary examples of non-generic resources are sensors and actuators. A node embedded into a lamp may offer an actuator allowing a program to switch the lamp on and off, as well as a sensor that makes it possible to determine the lamp's current state. Clearly, a node embedded into a refrigerator is likely to offer sensors and actuators of very different functionality. The environment described here is inherently heterogeneous: each node offers a unique collection of non-generic resources. These resources are critical, as they allow applications to interact with the physical world.

Nodes are deployed into the environment without any planning or particular order. Consequently, the programmer *cannot* assume any particular mix or topology of the nodes. A node configuration, although unknown, is quasi-static; it changes only occasionally (a new node may be added or an existing one may be moved to another location). Finally, the nodes are not supported by any infrastructure. They establish a system by forming an ad-hoc network.

The goal of this work is to develop a middleware layer for the above-described environment of unknown to the programmer, heterogeneous, resource-constrained sensor-actuator networks. Key postulated features of the new middleware, called ROVERS, are the following. It should largely free the programmer from the concept of the physical node. It should provide implicit resource discovery, convenient communications services, minimization of application-generated traffic, systematic (actually – ontology-driven) representation of sensors and actuators, as well as support for component-based programming. In the spirit of active sensor networks [1], it should be possible to freely deploy a new application on the resulting platform. Moreover, multiple, independent applications should be able to run concurrently. At the same time, the middleware should lend itself to an

This work was supported in part by the Polish Ministry of Education and Science, project no. 3 T11D 011 28.

implementation for a miniature operating system, like TinyOS [2].

The main contribution of this paper is a programming model satisfying the above postulates. The programming model, while offering an abstract view of the network, motivates a number of research problems of wider significance.

This paper is organized as follows. The ROVERS programming model is presented in Section 2. Application deployment is explained in Section 3. A programming example is provided in Section 4. Related work in the area of middleware for pervasive computing is referred to in Section 5. The paper is concluded and further work is outlined in Section 6.

## 2. ROVERS programming model

### 2.1. Logical structure of ROVERS application

A ROVERS application is a collection of lightweight, cooperating, concurrently running entities, called *micro-agents*. Micro-agents are organized into a hierarchical, tree-like structure (see Fig. 1.). The terms *"boss"* and *"subordinate"* will be used to denote a micro-agent's parent and child, respectively (in order to perform its tasks, the boss micro-agent uses results of the work of its subordinates). Usually, the leaves of the tree are micro-agents that interact with the physical environment by acquiring context information through sensors or effecting change through actuators. Higher-level micro-agents act as context synthesizers and decision makers.

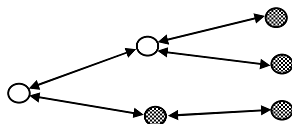

**Figure 1. A ROVERS application (micro-agents requiring special resources are in grey)**

A subordinate is created by its boss by means of a ROVERS primitive called *hiring*. At the beginning, only the root micro-agent exists. The root hires its subordinates. They, in turn, hire theirs, and so on. Hiring may occur at any time, possibly as a result of some specific context.

Once a subordinate is created, it can exchange messages with the boss through a dedicated, point-to-point, best-effort communications channel. The channel is set up and fully maintained by ROVERS. Addressing messages to the boss is implicit. Subordinates are addressed by ROVERS-assigned handles.

A boss and subordinate see each other through the latter's *micro-agent interface*. In fact, while hiring, the boss specifies the subordinate's interface, not implementation. An interface specifies what messages can be exchanged between the two. The subordinate sends *reports*, and the boss sends *commands*. For example, the interface of a temperature acquiring micro-agent may include a temperature report and a command to change the reporting period. Comprehensive libraries of reusable ROVERS micro-agents, implementing useful interfaces, can be developed.

Some micro-agents require special resources (sensors or actuators) to do their job. For example, the temperature acquiring micro-agent requires a temperature sensor. The sensor, however, may not be available on any physical node. In that case, the subordinate micro-agent will not be created by ROVERS, even though it has been hired. If it is too difficult for the boss to work without the subordinate, it may simply quit. If this process is repeated higher up the tree hierarchy, entire sub-trees may eventually be missing. The goal, however, is to make the application run even though some resources are not available. To compensate for the lack of knowledge about the deployment environment, micro-agents should be written so that they degrade their performance gracefully in case some of their subordinates are missing.

As follows from the above, the application tree is not fixed. It evolves as micro-agents hire their subordinates. It also depends on resources available in the environment.

The mindset of the ROVERS programmer is not node-centric; it is micro-agent-centric. The application is developed as a collection of communicating logical entities (micro-agents), some of which may interact with the world through sensors and actuators. As explained below, deploying and maintaining an application in an actual environment (discovering resources, allocating micro-agents to physical nodes, maintaining multi-hop communications channels, etc.) is done entirely by ROVERS.

### 2.2. Micro-agent internals

The ROVERS middleware adopts event-driven programming. The micro-agent is a collection of event handlers, each handling a specific *ROVERS event*. An event handler is a sequence of *ROVERS instructions*. Execution of a micro-agent amounts to occasional execution of one of its event handlers. All instructions are non-blocking; the time to execute a micro-agent's event handler is short and predictable. In between events, the micro-agent is inactive.

The ROVERS events and instructions include (a) typical programming constructs (e.g., a timer expiration event or a looping instruction), (b) micro-agent management primitives (hiring instructions), (c) micro-agent communications primitives (sending and receiving of a command or report), and (d) sensor and actuator representation primitives (e.g., a smoke detection event or an instruction yielding a reading from a temperature sensor).

All the events and instructions, except for the typical ones, constitute ROVERS primitives, summarized in Table 1. The notation and usage are exemplified in Section 4.

In order to derive sensor and actuator representation primitives in a systematic way, a target domain (e.g., home) is modeled by an ontology. An ontology preprocessor tool is used to generate a full (possibly huge) collection of instructions and events for the domain (samples are given in Table 1). Our initial work on this novel approach has been presented in [3].

**Table 1. Key ROVERS primitives**
(I/E – instruction/event, G/N – generic/non-generic)

| Name | Type | Functionality |
|---|---|---|
| **Micro-agent management** | | |
| ROVERS.hireGeneric | I,G | hire a generic subordinate |
| ROVERS.hireNonGeneric | I,G | hire a non-generic subordinate |
| ROVERS.getMicroAgent | I,G | get a handle to a hired subordinate |
| **Micro-agent communications** | | |
| Boss.*report* | I,G | send a report to the boss |
| *subordinate.command* | I,G | send a command to the subordinate |
| report *name* | E,G | report reception event |
| command *name* | E,G | command reception event |
| **Sensor and actuator representation (samples)** | | |
| Node.getTempInKitchen | I,N | get the kitchen temperature |
| Node.switchOnDeskLamp | I,N | switch a desk lamp on |
| Node.RefrigeratorDoorOpen | E,N | refrigerator door open event |

## 2.3. Non-generic primitives and micro-agents

The ROVERS events and instructions form a micro-agent execution environment provided by physical nodes; they can be thought of as building blocks for a virtual machine. Just as nodes differ in terms of their resources, so do their virtual machines. If an event or instruction is available on every node, it is *generic*; otherwise, it is *non-generic*. All the events and instructions, except for the sensor and actuator representation primitives, are generic. The latter are non-generic because each node has its unique set of sensors and actuators. On top of all the generic events and instructions, a node supports its own collection (usually only a handful) of non-generic ones.

If each event handled by a micro-agent is generic, and so is each instruction contained in any of its handlers, then the micro-agent is generic. Otherwise, it is non-generic.

A micro-agent can run on a node only if all events and instructions it uses are supported by the node's virtual machine. A generic micro-agent can run on any node. A node that provides all non-generic primitives needed by a non-generic micro-agent is called a *host* for this micro-agent. A non-generic micro-agent can run only on its hosts. The number of hosts is not known in advance to the programmer; it depends on a specific deployment environment. There may be more than one host, just one, or none at all.

The unknown number of hosts gives rise to the question of which of them to use when hiring a non-generic micro-agent. Currently, two programmer-selectable modes are included. In the *broadcast hiring mode*, an instance is created on every available host. In the *anycast hiring mode*, at most one instance (on a host picked by ROVERS) is created, no matter how many hosts are available. For each mode, the programmer should take into account the possibility that no instances are created at all (no hosts).

# 3. ROVERS application deployment

A ROVERS application may be preinstalled on a usual embedded node or on a minimal, stand-alone "application pill." Another option is to inject the application through a gateway node. In either case, there is a single physical node, called the origin, where the root micro-agent gets "hired" by ROVERS. Different applications may have different origins.

Starting at the origin, the running application may spread all over the environment. Each micro-agent may eventually run on a different node. Efficient micro-agent to physical node mapping is achieved through micro-agent mobility. Both generic and non-generic micro-agents are mobile, although for different reasons. Also, a different concept of code mobility is employed in each case.

The micro-agent to node mapping (micro-agent mobility) is in either case completely transparent to the programmer. Except `hire`, there are no code mobility primitives. All mobility-related decisions and operations are performed exclusively by ROVERS.

## 3.1. Non-generic micro-agent mobility

The reason for non-generic micro-agent mobility of is to move a micro-agent to a host (or all available hosts, depending on the hiring mode). When a non-

generic micro-agent is hired, ROVERS takes care of finding hosts, moving the micro-agent's code there, and instantiating the micro-agent (weak mobility). No mobility after that is allowed; the micro-agent runs on one node until it terminates.

When developing a non-generic micro-agent, the programmer is aware that its instance will be hosted by some physical node (i.e., a host). As explained below, this node-centric treatment of non-generic micro-agents is contrary to that of generic ones (which are node-indifferent).

Non-generic events and instructions can be thought of as services that a node offers. An application takes advantage of them through its non-generic micro-agents. Non-generic micro-agent mobility amounts to implicit (ROVERS-provided) service discovery.

### 3.2. Generic micro-agent mobility

By definition, a generic micro-agent can run on any node. When a generic micro-agent is hired, it is immediately instantiated. Initially, the subordinate and its boss run on the same node.

The reason for generic micro-agent mobility is (a) to reduce the total amount of traffic produced by the application or (b) to offload an overcrowded node. Traffic reduction is usually achieved by moving a boss closer (in terms of the number of hops) to its data-generating subordinates. When ROVERS decides that moving a generic micro-agent to a neighboring node would achieve any of the two objectives, the micro-agent is transferred there, along with its execution state (strong mobility). Thus generic micro-agents can be moved freely by ROVERS (based on the above, non-functional criteria). Any node-related notion (e.g., location) is not applicable to them. The programmer does not think of nodes when developing generic micro-agents.

## 4. ROVERS programming example

A simple kitchen-related application has been developed to illustrate the ROVERS programming model. Its job is to detect smoke and turn on a fan, if needed. The tree is shown in Fig. 2.

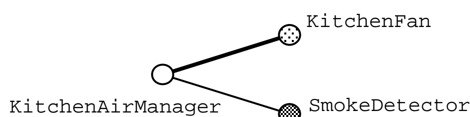


**Figure 2. The kitchen application tree**

As can be seen, the generic root micro-agent, `KitchenAirManager`, hires two non-generic subordinates: `SmokeDetector` and `KitchenFan`. The former is hired in the broadcast mode (as denoted by the thick line), and the latter in anycast.

The source code for the entire application is presented in Figs. 3-5. It is written in our informal notation called "ROVERS C," inspired by the nesC programming language [2]. The ROVERS primitives and ROVERS C keywords are in bold. Many items have a prefix enhancing readability, e.g., `Boss.` (the report sending primitive), `Node.` (non-generic events or instructions), or `ROVERS.` (some generic primitives).

Micro-agent interfaces for the subordinates are given in Fig. 3. The smoke detector sends a report alerting about high smoke level. The fan micro-agent sends a report alerting that the fan is on and accepts commands to turn the fan on and off. The fan interface also includes a report and command for establishing boss-subordinate communications (`ping()` and `pong()`), explained below.

```
interface ISmokeDetector{report smokeLevelHigh();}
interface IKitchenFanManager {
 report fanOn(), ping();
 command turnOn(), turnOff(), pong();
}
```

**Figure 3. Micro-agent interfaces**

The smoke detector micro-agent implementation is given in Fig. 4. The implemented interface is declared with the `delivers` keyword. The micro-agent contains three event handlers. A node's smoke sensor is represented by two non-generic events: `Node.smokeAlertOn` and `Node.smokeAlertOff`. The fact that these events are used makes the micro-agent non-generic and insures that ROVERS will instantiate it only on nodes equipped with a smoke sensor. No non-generic instructions are used by the micro-agent. In the handler for the generic timer expiry event, a high smoke level report is conditionally sent to the boss. This is the only way this micro-agent communicates with its boss, as declared in the `ISmokeDetector` interface.

```
microagent SmokeDetector delivers ISmokeDetector {
 int alert = FALSE;
 event timer{if(alert == TRUE) Boss.smokeLevelHigh();}
 event Node.smokeAlertOn{alert = TRUE;}
 event Node.smokeAlertOff{alert = FALSE;}
}
```

**Figure 4. The SmokeDetector micro-agent**

The kitchen fan micro-agent implementation is given in Fig. 5. Four events are handled; three of them are communications events (command receptions). In the timer expiry handler, the fan status is reported.

```
microagent KitchenFan delivers IKitchenFan
{
 event timer {
  if(!ROVERS.gotCommand()) Boss.ping();
  if(Node.getKitchenFanStatus() == ON) Boss.fanOn();
 }
 command turnOn() {Node.turnOnKitchenFan();}
 command turnOff() {Node.turnOffKitchenFan();}
 command pong(){}
}
```

**Figure 5. The KitchenFan micro-agent**

A node's fan switching actuator is represented by the three non-generic kitchen fan instructions. Consider, for example, `Node.turnOnKitchenFan()`. Note that the instruction specifies not only an object the node is embedded in (a fan), but also the logical location of the object. The instruction has been derived from a home domain ontology that included logical locations. ROVERS will instantiate the micro-agent only on nodes equipped with a fan switch *and* located in a kitchen.

The `KitchenAirManager` root micro-agent is given in Fig. 6 (its logic has been greatly simplified for brevity). The interfaces of hired subordinates are declared with the `hires` keyword (of course, in a general case, a micro-agent can both deliver and hire). As can be seen, the boss deals only with the interfaces of its subordinates, not their implementations.

```
microagent KitchenAirManager
 hires ISmokeDetector, IKitchenFan
{
 int smokeFreePeriods = MAX_PERIODS;
 ROVERS.MicroAgent fanManager = NULL;
 event once{
  ROVERS.hireNonGeneric(ISmokeDetector,BROADCAST);
  ROVERS.hireNonGeneric(IKitchenFan,ANYCAST);
 }
 event timer{
  if(smokeFreePeriods < MAX_PERIODS){
   fanManager.turnOn();
   smokeFreePeriods++;
  }
 }
 report ISmokeDetector.smokeLevelHigh(){
  smokeFreePeriods = 0;
 }
 report IFanManager.fanOn(){
  fanManager = ROVERS.getMicroAgent();
  if(smokeFreePeriods == MAX_PERIODS)
   fanManager.turnOff();
 }
 report IFanManager.ping(){
  fanManager = ROVERS.getMicroAgent();
  fanManager.pong();
 }
}
```

**Figure 6. The KitchenAirManager micro-agent**

The `once` event, known from the Maté virtual machine, is generated by ROVERS when a micro-agent is instantiated. In the `once` event handler, the smoke detector and kitchen fan micro-agents are hired, in the broadcast and anycast mode, respectively. The application detects smoke using all available smoke sensors, no matter where located. A single fan located in the kitchen is turned on if smoke is detected.

The handlers for timer expiry and three communications events (report receptions) follow. The micro-agent turns the fan off if no smoke has been detected by any of the sensors for a number of timer periods.

The `fanManager` variable is intended to store the handle for a subordinate. The handle is needed to identify a subordinate for different purposes, including sending a command. The handle can be retrieved by the `ROVERS.getMicroAgent()` primitive only while handling a report from the subordinate. The exchange of the `ping()` and `pong()` messages of the `IKitchenFan` interface ensures that the kitchen air manager obtains the handle for the kitchen fan.

If a command is sent with an un-initialized micro-agent handle, the sending instruction is quietly discarded by ROVERS.

A deployment example for the `KitchenAirManager` application is presented in Fig. 7.

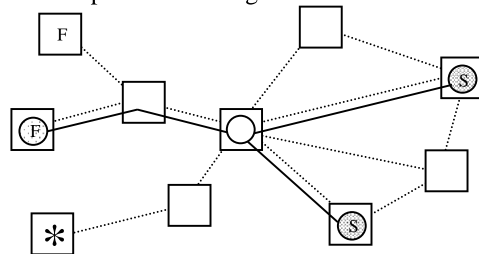


**Figure 7. A deployment of KitchenAirManager**

The origin (the physical node through which the application is injected, and the root micro-agent is instantiated) is marked with the asterisk. As can be seen, ROVERS has instantiated the non-generic micro-agents (the grey circles) at their hosts (F – fan, S – smoke detector), according to their hiring modes. Also, ROVERS has detected that it pays to move the generic root (the white circle) towards smoke sensors to minimize traffic generated by the application. This illustrates non-generic and generic micro-agent mobility, respectively.

## 5. Related work

Middleware support for open pervasive computing and active wireless sensor networks is a dynamic research area. A number of middleware layers comparable to ROVERS have been described in the literature. These systems include Maté [4], SensorWare [5], Agilla [6], SmartMessages [7], DFuse [8],

MagnetOS [9], Solar [10], PIECES [11], Deluge [12], Impala [13], among others.

In spite of a large number of competing systems, we believe that ROVERS offers the programmer a unique abstraction of a sensor-actuator network. To the best of our knowledge, none of the above systems offers the following in a single, coherent package: (1) micro-agent-centric (not node-centric) programming, (2) implicit resource discovery (viz. non-generic micro-agent mobility), (3) convenient communications services (viz. communications channel with a logical addressing scheme), (4) implicit minimization of application-generated traffic (viz. generic micro-agent mobility), (5) ontology-driven representation of sensors and actuators, and (6) support for component-based programming (viz. micro-agent interfaces). We justify this claim with a couple of examples.

In Maté, there is no support for a distributed application built of heterogeneous agents. Maté is primarily a system to generate application-specific, but homogeneous virtual machines. Moreover, node-centric view of Maté-based applications differs significantly from our approach.

In Agilla, there is no notion of a distributed application; agents are injected into a network separately, and different agents are not functionally coupled (as in ROVERS). Agilla agents are addressed by location (not by attributes); thus the system is not aimed at location-unaware nodes. Also, in Agilla, agent mobility has to be handled by the programmer.

In SensorWare, there is no implicit resource discovery. A SensorWare script has to explicitly inquire about the presence of a sensor or actuator. Moreover, just as in Agilla, script mobility is under the programmer's control.

In systems like Deluge and Impala, the main feature is efficient code propagation, and not full support for the active network paradigm.

Quite a few of the competing systems (e.g., PIECES, SmartMessages, Solar, MagnetOS) have been implemented on resource-rich, PDA-like platforms. Our experience in resource-constrained systems indicates that ROVERS mechanisms are lightweight enough to be implementable on TinyOS-like nodes.

## 6. Conclusions and further work

The ROVERS middleware offers a simple and abstract programming model for heterogeneous sensor-actuator networks. The programming model, in turn, gives rise to meaningful research problems. They include protocols for micro-agent mobility. In particular, non-generic micro-agent mobility requires protocols for attribute-based addressing (non-generic events and instructions supported by a node can be thought of as the node's attributes). Relatively little work has been done in this area. An original idea included in the programming model and requiring further work is the derivation of lightweight programming artifacts (events and instructions) from an ontology [3]. More work is also needed on how to better support writing pervasive computing applications for an unknown mix of nodes.

We are currently working on all of the above problems. Each of them is apparently meaningful in its own right. In addition, we are implementing the system. While the programming model is not tied to any specific architecture, the TinyOS platform has been picked as the implementation testbed.

## 7. References


[1] Levis, P. and Culler, D, "Active Sensor Networks", in Proceedings of NSDI 2005, Boston, USA, 2005

[2] Gay, D., et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems", in Proceedings of PLDI 03, San Diego, California, USA, 2003

[3] Domaszewicz, J. and Rój, M., "Lightweight Ontology-driven Representations in Pervasive Computing," in Proceedings of NCUS 05, Nagasaki, Japan, Springer, 2005

[4] Levis, P. and Culler, D., "Maté: A Tiny Virtual Machine for Sensor Networks", in Proceedings of ASPLOS-X, San Jose, CA, USA, 2002

[5] Boulis, A., Han, C.-C., and Srivastava, M.B., "Design and Implementation of a Framework for Efficient and Programmable Sensor Networks", in Proceedings of ACM MobiSys 2003, San Francisco, California, USA

[6] Fok, C.-L., Roman, G.-C., and Lu, C., "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications", in Proceedings of ICDCS 2005, Columbus, Ohio, USA, 2005

[7] Kang, P., et al., "Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems", *The Computer Journal*, 2004. 47(4): p. 475-494.

[8] Kumar, R., et al., "DFuse: A Framework for Distributed Data Fusion", in Proceedings of ACM SenSys 2003, Los Angeles, CA, USA, 2003

[9] Barr, R., et al., "On the Need for System-Level Support for Ad Hoc and Sensor Networks", Operating System Review, 2002. 36(2): p. 1-5.

[10] Guanling, C., Li, M., and Kotz, D., "Design and implementation of a large-scale context fusion network", in Proceedings of MobiQuitous'04, Boston, USA, 2004

[11] Liu, J., et al., "State-Centric Programming for Sensor-Actuator Network Systems", *IEEE Pervasive Computing*, 2003. 2(4).

[12] Hui, J. and Culler, D., "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale", in Proceedings of SenSys'04, Baltimore, MD, USA, 2004

[13] Liu, T. and Martonosi, M., "Impala: a middleware system for managing autonomic, parallel sensor systems", in Proceedings of PPoPP'03, San Diego, California, 2003