Distributable lightweight components in a resource-aware ubiquitous computing environment

Peter Rigole
Yolande Berbers
Tom Holvoet

Report CW 352, October 2002

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Distributable lightweight components in a resource-aware ubiquitous computing environment

Peter Rigole
Yolande Berbers
Tom Holvoet

Report CW 352, October 2002

Department of Computer Science, K.U.Leuven

Abstract

This report presents an innovative project about the development of a middleware layer for ubiquitous computing environments using a component-oriented approach. Ubiquitous computing systems tend to have special properties, such as resource-awareness, resource harvesting (cyber foraging) and application relocation, requiring them to use technology such as wireless communication, service discovery and resource-aware operating systems. This paper describes how the project integrates these properties into an embedded component system.

Keywords : embedded systems, ubiquitous computing, component-oriented programming.
CR Subject Classification : C, 2, 4
CR Subject Classification : D, 2, 10
CR Subject Classification : D, 2, 11

Distributable lightweight components in a resource-aware ubiquitous computing environment

Peter Rigole, Yolande Berbers, Tom Holvoet
Department of Computer Science; KULeuven; Celestijnenlaan 200A, B-3001 Leuven
+32 16 327640; {peter.rigole,yolande.berbers,tom.holvoet}@cs.kuleuven.ac.be

Abstract
This paper presents an innovative project about the development of a middleware layer for ubiquitous computing environments using a component-oriented approach. Ubiquitous computing systems tend to have special properties, such as resource-awareness, resource harvesting (cyber foraging) and application relocation, requiring them to use technology such as wireless communication, service discovery and resource-aware operating systems. This paper describes how the project integrates these properties into an embedded component system.

Keywords
embedded systems, ubiquitous computing, component-oriented programming

INTRODUCTION

The impressive evolution of computer architectures over the past decade has led to more powerful and even smaller computing devices of many kinds. In particular the world of embedded systems has eagerly embraced this technological development since it allows the use of modern software design methodologies on embedded devices, instead of the usual low-level software design. One of these methodologies is component-oriented programming (see [16, 13]).

One particular research domain was actually waiting for this evolution. This is the domain of ubiquitous computing, also often referred to as pervasive computing. The idea behind pervasive computing is to make the computing power disappear in the environment so that it becomes invisible, but is still always there whenever needed. New applications for this invisible technology are invented every day in the literature [17, 15, 14, 11], but few software architectures have yet been built to support any such application. One of the reasons for this is the many properties these systems need in order to be successful. The ability to be resource-aware and to extend the system resources of an embedded device to the resources it can find in its environment are the most important ones. These properties require wireless ad hoc networks.

The purpose of the proposed project is to develop a general software environment that supports component-based applications for embedded systems within the context of ubiquitous computing. One of the problems the project intends to solve is management of the system resources of the component system. This component system is the host platform for components, which are the building blocks constituting the entire application. These components are linked together by their ports through which they communicate. Two ports can only be connected if their interfaces are compatible (if handshaking is successful).

The system resources of an embedded device running the component system can be extended by addressing the resources on other devices in the immediate environment; this is called Cyber Foraging in the literature (cf. [15]). This addressing can take place over a wireless connection between nearby devices. Using system resources on remote devices can be done either by directly addressing them through the connection with a remote component (which has access to the resource), or by moving a local component to the other system so that execution in this component is handled on that system (CPU cycle resource) and its memory is allocated there (memory resource) as well. Other resources taken into

account include bandwidth, battery power, class-availability and system-specific resources such as a printer or a sound card.

Our paper is further organized as follows. The first section describes the resource-awareness property of the proposed system and the methodology that can be used for its implementation. The second section covers the discovery procedure which is needed to enable ad hoc networks to exchange services, followed by a discussion of the strategies needed for component relocation. Thereafter, an overall view of the design of the system is given, followed by a brief conclusion.

A RESOURCE-AWARE COMPONENT SYSTEM

The first and most important property of the proposed component system is its resource awareness. The system needs this resource awareness to ensure the reliability and availability of its active software components. The system should, in fact, try to comply with the dynamically varying demands of its applications in any possible way, using every resource it can harvest locally or remotely on a device that forms part of a local ad hoc network. Due to the limited resources of small embedded devices and the growing need to use them as efficiently as possible, resource awareness should be part of the context-consciousness of the middleware layer that supports ubiquitous computing. In the literature, a great deal of research activity can be found in the area of context-aware environments which often overlap with the domain of mobile agents, such as [12] which describes context-aware agents in ad-hoc networks. Furthermore, existing resource monitoring systems are discussed in [9] by A. Acharya, and an assessment is made in [10] by F. Chang about applications that adapt their behaviour dynamically to the availability of the system resources so that they do not exhaust them.

The six important resources we are considering in our system are:

Bandwidth: the available bandwidth between two physically separated components which are remotely linked via a wireless connection

Memory: each component needs some memory, partly to store its objects and program code, and partly to build up a stack to process the messages it receives from other components

CPU power: some processing power, provided by the CPU

Class-availability: a component deploys either objects of custom-made types (classes) or it can use classes from standard Java libraries. Some embedded devices, however, only provide a limited set of libraries (a device without a sound system, for example, does not need audio libraries)

Battery Power: since embedded devices are often mobile, many of them exist on their own batteries. Using these batteries economically is thus the utmost importance for these devices

Special system resources: these are extra resources that are provided only by some specific devices. A video projector, for example, offers a way of projecting images on a screen. This can be regarded as a special system resource.

The consumption of these resources by each of its components has to some extent to be measured by the component system. For this reason, reliable support from the underlying operating system is needed. We assume that we have an operating system available that provides the component system with a fixed amount of these resources during its life time. The component system thus becomes a virtual operating system that hosts components and that has a constant level of resources at its disposal to assign to its components. This is the only reliable way to control the consumption of resources and make commitments to the components.

Given this assumption, we are able to monitor each component's consumption of the given resources. The use of a component's **bandwidth** is easy to track since all communication between components, even local communication, passes through the component system. The average size and the number of messages sent per time unit gives an idea of the component's required bandwidth over a specific port.

The focus of our research will undoubtedly be on the behaviour of components with respect to their **memory** allocation, since memory is generally seen as the major obstacle to the evolution of computing systems, because of its slow speed and power consumption. Our view distinguishes between two memory areas per component. One is used by the passive component to store objects, class definitions, method code, class variables and symbolic link information. Its size defines a component's size (Component Memory, or CM) on the system. The other section is allocated when a message has to be processed (Message Memory, or MM) by a component. The thread that goes with the component

will then build up its stack, where increase in size is only dependent on the algorithm that processes the message. The size of this stack should be limited by the size of the Message Memory of the component.

The minimum **CPU power** required by a component to process incoming messages rapidly enough is hard to express in a simple measurement. The number of CPU cycles required to process one message will vary wildly among different architectures (especially RISC versus CISC) which would make it a useless criterion for comparison. The idea in this research project is to work with heuristic methods based upon reference values of the component's required CPU power on certain architectures. CPU-scheduling research for solving timing-constraint problems between components is of lesser importance for us because of the potentially unreliable and unpredictable speed of the wireless network. Simple timing constraints between two ports will, however, be implemented and monitored by the system.

Available **battery power** can be queried via the operating system. An almost exhausted battery should put the component system into a safe mode, in which the system becomes extremely careful when allocating power consuming resources by accepting only the most urgent components, or by moving some of its components to other devices in the environment. It should be aware that moving components to a remote location increases the effort and thus the cost (power needed) of communicating with it.

Class-availability can easily be checked by Java's class loader. A component should automatically be refused by the system if the required Java libraries are not available.

Finally, the **special system resources** require a device-specific solution that has to be implemented in the form of a common component with extra powers. The device with the extra system resource (a sound system for example) has to provide access to this resource through this special component.

Methodology

Implementing the monitoring ideas mentioned above is no piece of cake. Fortunately we can seek assistance from advanced techniques such as real-time (RT) operating systems, native interfaces, sophisticated profilers, etc.. Real-time Linux is the obvious first choice operating system since it is known as a host platform for real-time Java (RTJ for short); the specification of the RTJ is available at [7]. The built-in techniques that allow resource-aware programming are extremely useful for memory monitoring. Real-time Java offers an innovative way of handling different kinds of memory, such as immortal memory, physical memory and scoped memory; it can confine memory to a maximum size, and much more besides. This makes it an appropriate environment for designing a memory constraint monitor, but it can also help us access other system information such as battery power, the total amount of available bandwidth or other special resources (an attached screen for example).

An alternative monitoring solution is presented by the Java Virtual Machine Profiling Interface (JVMPI, see [4]). Implementing of this interface can extract an incredible amount of data from the operational Java virtual machine. Information about the memory used as well as about the activity of the different threads (CPU usage) can be made available by implementing the profiling interface and can therefore serve our component system.

Another technique would be to implement Java Native Interface (JNI, see [2]) wrappers to call on native library functions. These can be deployed to query and control certain resources like battery power, CPU usage, some special system resources (such as a sound driver), etc..

DISCOVERY

To meet the requirement of Cyber Foraging, the component system has to be able to find other component systems and their components in its local environment in the first place. There are many environments on the market that provide the functionality to localize services on a standard computer network. Examples are Jini [5], Corba [1] and UPnP [8]. Since we are aiming for wireless communication, another approach needs to be taken, but concepts from the above examples, concerning their functionality and implementation aspects, should be borne in mind when developing our own system. One wireless communication protocol seems extremely suited to an embedded ubiquitous environment since it is economical when it comes to power consumption and its range of ten meters looks perfect for local ad hoc networks. Its name is Bluetooth.

Bluetooth takes care of a great deal of the discovery procedure by itself. Its standard protocol learns which other devices are within range and the Bluetooth Service Discovery Protocol (SDP) is able to query their available services.

Queries which look useful for our component system are:

- find a component system based on some required system resources (such as available bandwidth, memory, special resources, e.g. a printer, etc..);
- find a component based on its name;
- find a component based on a port (therefore based on an interface);
- find a component based on the kind of functionality it offers (for example components controlling a home automation system).

Once the desired (active) component has been found on a remote component system, all that needs to be done before it can be used is to wire it to the requesting component over the remote link.

COMPONENT RELOCATION STRATEGIES

Component relocation means moving an operating component to another component system in order to place it in a richer environment (which has more resources to offer), or to free the resources it consumes on the local system. The ability to relocate parts of an active application is a primary requirement in order to make Cyber Foraging possible.

The key question is now: "How should the component system decide to move one or more of its components?". The literature describes some application scenarios in ubiquitous computing, such as in the work of Mark Weiser [17, 18] and Satyanarayanan [15]. They all have rather futuristic visions that go far beyond the awareness of limited resources. Their vision is an entirely context-aware system that cooperates proactively to help the user. They anticipate the user's actions and try to make his tasks as comfortable as possible using the computing power that is located in the user environment. Alternatively, adress the computing power that will be in the user's environment within some minutes, since the system takes the location of the user into account as well. The relocation of active software is simply a means of achieving this. The fact that there are so many other such futuristic ideas attached to ubiquitous computing makes us realize how much impressive research still remains to be done in this field. Our component system ought to represent one first step in the right direction.

Strategy

The system we describe here only focuses on the relocation of components in the environment due to a lack of one of the six system resources mentioned. The decision is made autonomously by the system whenever it deems it necessary. So, this phase of the project boils down to developing strategies that wisely determine when to relocate one or a group of components based on the resource consumption and resource availability of the system.

There is no need to calculate an optimum solution, given the available resources in the environment and the needs of the active components. This computation would, first of all, take too much time but, in addition, the solution would probably be completely useless since it may require unfeasible reorganization of the components. Relocating components takes time and is a costly operation for resources (use of extra bandwidth, CPU, etc.), which also has to be taken into account.

The job of the component system consists of meeting the resource needs of its components by replacing them to an appropriate system, without causing an excessive increase in bandwidth (Bluetooth has a practical bandwidth of about 434 Kb/sec, both upstream and downstream). This is why it should have an algorithm that first selects a group of components, leading to a feasible increase in bandwidth when they are relocated and, secondly, looks around for a host that is willing to accept the selected components. Graph-theory, linear programming theory [19] (the Simplex and Branch-and-Bound method, the Knapsack problem, etc.), or a declarative programming language such as Prolog might help to find a solution.

DESIGN OF THE COMPONENT SYSTEM

We chose to use Java to implement the proposed component system because of its platform-independence, which Cyber Foraging needs in order to move components between component systems on different architectures. Two light-weight Java virtual machines designed by Sun Microsystems are available and can be used in an embedded environment. They can be compiled for any desired architecture since the source code is available as well. Another option is, of course, to use an implementation of the real-time Java virtual machine.

A component has to be considered an entity that has its own code and data space and acts reactively to messages from its neighbouring components. It is a reusable and documented building block for component-oriented software design. Components can be linked together by their ports (interfaces)

into a cooperative composition. The component system is then the complete layer that provides the environment for components to live in. It loads components, it glues components together, it makes sure the components can process their messages, it maintains remote links to other component systems, etc.. Messages between components are always sent asynchronously, which has the advantage that threads never have to wait for a reply. This is definitely advantageous during wireless communication. Synchronous messages are, in addition, much more likely to cause deadlocks, which are hard to detect or prevent with resource-constraint embedded devices.

The component design is supported by new syntactical constructions and keywords, which are transformed into component system calls during an extra compilation step.

Resource-awareness

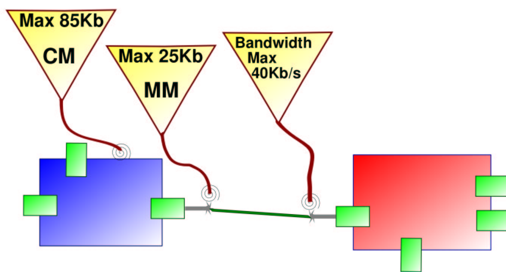

Figure 1: Components and their resource profiles

The component system is not just aware of the resources the components use. It also has an idea of their maximum resource consumption, so that it can make predictions about the quantity of the available resources. In order to obtain this information, each component has to reveal a resource profile containing the maximum quantity it needs of each resource. Figure 1 shows two components and some of their resource profiles. The left-hand component gives the maximum size of its component memory as well as the maximum size of its message memory. The second component reveals the maximum bandwidth of one of its ports (40Kb/sec). The component system can refuse to activate any component that requires too much of its resources and it should punish a component that does not maintain its profile.

Java and Bluetooth

Mobility is a core requirement of ubiquitous computing, as is wireless communication with other devices in the environment. Bluetooth is the technology we chose from the available wireless protocols to build this prototype component system. Making Java interact with it means implementing the Java API specification for Bluetooth (see JSR-82, [3]). This implies that we have to address the Bluetooth protocol stack by calling on native libraries. This is supported in Java via the Java Native Interface (JNI, cf. [2]). The Bluetooth protocol stack we have in mind is the open source BlueZ ([6]) protocol stack. The Bluetooth service discovery protocol should then be moulded and extended to implement our own discovery requirements.

Relocation

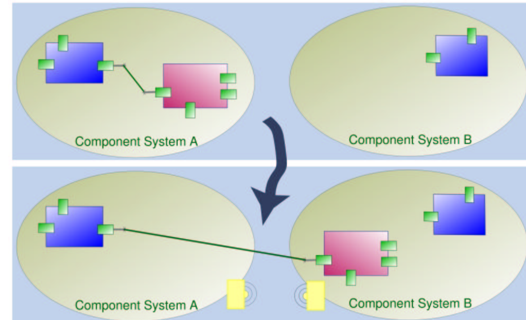

Figure 2: Component relocation

Technically achieving the relocation of active components (see the illustration in Figure 2) requires certain steps to make it a safe procedure. A simplified description of what should happen is as follows. First of all, the component should be interrupted in a secure state (when it is not processing messages) and its internal state should be extracted and stored. The next step consists of transmitting the component's code and stored state over the wireless network to the other component system. On the other side, the new host then revives the component by loading its classes and initializing the component using its internal state. After rewiring the ports of the relocated component (some of these links are now transmitted over the wireless beam), the component can be activated again and a completion signal should be sent to the old system so that it knows when to deallocate all the resources which the relocated component was using.

In addition to this technical phase, the relocation strategies developed (see section) should be moulded into an implementation that is even functional on small embedded environments. Furthermore, the system should handle the problems caused by glitches in the wireless network, or even the complete disappearance of the network (for ex-

ample when a device gets out of range of the ad hoc network) in order to underscore the robustness of the application.

CONCLUSION

This project owes its relevance to the broad spectrum of applications waiting for their implementation in a ubiquitous computing environment. It shows how the basis for a ubiquitous computing system could be designed using an embedded component-oriented approach. These are precisely components that make the requirements of resource foraging, application relocation and resource-awareness achievable, thanks to their fine-grained nature.

Although much work still remains to be completed in this growing research domain, the challenging vision of many of its researchers encourages the ongoing search for a world offering ubiquitous computing power, one that hides its physical appearance in our daily environment.

References

[1] Corba - omg. http://www.corba.org/.

[2] Java native interface. http://java.sun.com/products/jdk/1.1/docs/guide/jni/.

[3] Java specification request for bluetooth. http://jcp.org/jsr/detail/82.jsp.

[4] The java virtual machine profiling interface (jvmpi). http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html.

[5] Jini - sun microsystems. http://wwws.sun.com/software/jini/.

[6] Linux bluetooth protocol stack. http://bluez.sourceforge.net/.

[7] Real-time java (rtj). http://www.rtj.org/.

[8] Universal plug-and-play forum. http://www.upnp.org/.

[9] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[10] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *HPDC*, pages 11–20, 2000.

[11] S. A. N. Shafer G. D. Abowd, B. Brumitt. At home with ubiquitous computing: Seven challenges. pages 256–272, Atlanta, GA, September 2001. Ubicomp 2001, LNCS 2201.

[12] Amy Murphy Gruia-Catalin Roman, Christine Julien. A declarative approach to agent-centered context-aware computing in ad hoc wireless environments. Orlando, Florida, USA, May 2002. In the workshop of the International Conference on Software Engineering.

[13] Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *Proceedings, International Component Users Conference*, Munich, Germany, 1996. SIGS.

[14] D. Salber, A. Dey, and G. Abowd. Ubiquitous computing: Defining an hci research agenda for an emerging interaction paradigm: Tech, 1998.

[15] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, August 2001.

[16] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, 1998. ISBN 0-201-17888-5.

[17] Mark Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–10, September 1991.

[18] Mark Weiser. The world is not a desktop. *ACM Interactions*, pages 7–8, January 1994.

[19] Wayne L. Winston. *Operations Research.* Duxbury Press, third edition, 1993.