A Component-Based Infrastructure for Pervasive User Interaction

Peter Rigole[1], Chris Vandervelpen[2], Kris Luyten[2], Karin Coninx[2], Yolande Berbers[1], and Yves Vandewoude[1]

[1] K.U.Leuven, Department of Computer Science
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{peter.rigole, yolande.berbers, yves.vandewoude}@cs.kuleuven.ac.be
[2] Limburgs Universitair Centrum
Expertise Centre for Digital Media
Universitaire Campus
B-3590 Diepenbeek, Belgium
{chris.vandervelpen, kris.luyten, karin.coninx}@luc.ac.be
http://www.edm.luc.ac.be

Abstract. Since a growing number of different mobile computing devices are used in pervasive and ubiquitous environments, the need to adopt new approaches for designing and implementing pervasive interactive software with minor effort is emerging. In this paper we present a process that facilitates the design of next-generation interactive software for pervasive environments. We created a distributed runtime infrastructure that enables the distribution of software components on heterogeneous, networked and embedded hardware systems. Some of these components or compositions of components will require interaction by human users from a large range of different devices. To make the deployment of consistent and functional User Interfaces in these pervasive environments easier, Interaction Components are introduced into the runtime infrastructure which enable the presentation of component and service behavior to human users.

1 Introduction

According to Mark Weiser a ubiquitous computing environment is an environment consisting of heterogeneous systems that interact with each other and with human users in a transparent way [29]. The dynamic nature of this kind of environments raises the need for more flexible methods to construct and design interactive applications. We propose a Component-Based Software Development (CBSD) [22] approach that supports dynamic human interfacing capabilities for exposing component functionality towards the user.

One of the major problems for the design of interactive applications in pervasive computing environments is the diversity of the available interfacing hardware and the constraints imposed by this hardware. These systems differ in interaction

possibilities such as screen size, availability of audio, keyboard or stylus input, etc.. It is important to take these constraints into account when designing and implementing components for this type of constrained devices.

Our solution proposes an indirect exposure of a component's interfacing needs through Interaction Components (ICs). This way, interfacing needs can be described in an abstract way at the level of a component, and transformed into concrete interfacing widgets at runtime by the ICs. In addition, this approach also tackles the problem of runtime user interface mobility between devices offering heterogeneous interfacing capabilities. During a relocation of software components from one host to another, the concrete user interface is regenerated using the abstract interface representation and the limitations of the new host.

Dynamic mobility of application functionality between heterogeneous devices is supported by the middleware layer that incorporates our component framework. Our component architecture has been designed for use in environments where flexible component mobility is often needed. It is therefore extremely suited to be deployed in conjunction with Interaction Components.

The main goal of our approach is twofold and can be summarized as follows:

- Provide a component runtime infrastructure which enables the easy creation of distributed, interactive applications for heterogeneous computing environments;
- Provide an easier way to manage consistent user interaction in heterogeneous environments;

In section 2 we give an overview of related work in this research domain. Section 3 introduces the middleware we created to support our main goals. We continue by pointing out the process we use to provide pervasive user interaction in section 4. Section 5 shows how the infrastructure gets the work done in real life situations by discussing a case study we worked out to verify our approach. We elaborate on problems we would like to tackle in the near future in section 6 and conclusions are given in section 7.

2 Related Work

Ponnekanti et al. ([17, 18]) describe a service framework for ubiquitous computing called ICrafter. Its main goal is to provide flexible interaction with services that are present in an interactive workspace. To enable this, ICrafter makes use of an infrastructure centric Interface Manager (IM) from which user appliances in the interactive workspace can request UIs for registered services or compositions of services. Such a request results in the selection of a generator for that service(s). This selected generator then sends UI markup needed for interacting with a particular service back to the user appliance. This approach is different from ours in which a HLUID (High Level User Interface Description) of the UI is directly attached to the interactive components without the use of a central Interface Manager.

Hodes et al. [10] describe the notion of universal interaction. This concept allows a device (the universal interactor) to adapt its functionality to use newly discovered services when the user moves to new environments. It embodies *transduction protocols*: these protocols map functionality to a UI suitable for representing it and it takes into account the portable device that is used as "remote control".

The Jini [15] framework is a service architecture that provides service discovery mechanisms in order for services to be able to locate each other. The Jini API enables one to easily create and deploy new services and clients that can use these services. The usual way of adding user interfaces to Jini services is by adding attributes bearing UI-code that can be instantiated on other hosts. The ServiceUI [21] project, however, has defined a standard approach for attaching UIs to Jini services.

OSGi [2] is an open services platform similar to the Jini framework, although OSGi wants to be more generic by offering bridging functionality between different kind of devices (such as Jini devices, UPnP devices, HAVi devices, etc.). OSGi often uses the concept of a component, also called a bundle, which provides one or more services. These components can discover and query each other for finding their required functionalities. They have no notion of connectors for indirect communication like in other component based systems (such as Fractal [4] and the SEESCOA[1] component system), but they directly address each other's interfaces and know exactly which other components they communicate with. Unfortunately, as Jini, OSGi offers no specific support for a presentation layer.

3 Supporting Middleware

From the ISTAG[2] AmI requirements [9], we may conclude that ambient intelligence middleware is bound to become flexible in nature for several reasons. First, the applications it supports demand a robust platform that allows for dynamic adaptability and extensibility. The shape of an AmI application is not static but is more or less amoebic, meaning that it can adapt to the environment and move around among interconnected systems. The advantages of reflective middleware [7] to enhance this advanced adaptive behavior are definitely required. Second, the middleware must be available on a broad spectrum of hardware. Be it ultra-small embedded devices or high-end embedded devices, the middleware must always be present to support its applications.

The previous requirements lead to a micro kernel [20] structure as the basis for AmI middleware. The core of the architecture is very small and only includes the basic functionality to support the most primitive applications. Additional features such as distribution, mobility, contracting and monitoring can be added to the runtime system's core. Such micro kernel structure permits to compose the

[1] described further in this paper
[2] The European Information Society Technologies Advisory Group

runtime environment according to the needs and the abilities of the underlying hardware.

According to the ISTAG report, approaches for integrating both parts of the AmI view need a fluent integration of user interfacing with distributed computing systems. Natural interaction has to be grafted onto the domain layer, maintaining properties such as heterogeneity, mobility and distribution. The symbiosis between a flexible interaction infrastructure and a powerful middleware layer leads to pervasive user interaction. In this section, we further elaborate on how the DRACO component framework supports the SEESCOA component methodology.

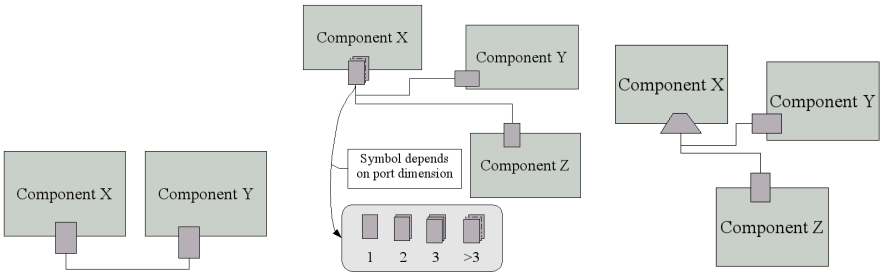
3.1 The SEESCOA Methodology

The SEESCOA methodology is a component-based software development methodology that aims at developing robust applications for high-end embedded systems. Concepts as component, port, contract and connector are the first-class entities that are used for building applications. *Components* define the level of granularity and are strictly delineated and reusable functional entities of which the applications are composed, *Ports* define the interfaces towards a component that can accept asynchronous messages from other ports, *Connectors* are used for setting up a communication link between two ports and *Contracts* define the specification of a component or a port on four levels: a syntactical level, a semantical level, a synchronization level and a Quality of Service (QoS) level. The synchronization level specifies the order in which messages are allowed to be sent between ports and the QoS level defines several Quality aspects that are related to the component or the port. Examples are message timing issues, memory usage, bandwidth usage, etc.. SEESCOA has three types of ports: *Single Port*, *Multiport* and *Multicast Port*. Connected components are represented in this paper as in figure 1. Further references to the SEESCOA methodology can be found in [24–26, 19, 28].

The SEESCOA methodology is appropriate for deployment of distributed components for two major reasons. First, event-driven operation by allowing only asynchronous messages is efficient in distributed environments where communication delays are realistic. Remote synchronous communication means a thread is waiting for a reply before continuing its task, which implies inefficient distributed cooperation. Second, the inherent granularity of a component composition makes a component an ideal migration unit. Relocating mobile components when required by environmental conditions is essential to provide the flexibility needed by AmI.

3.2 The DRACO Runtime Environment

DRACO is a Java-based micro kernel runtime environment for pervasive computing applications that has been developed as a proof of concept and testbed environment. The core of the DRACO system only supports the basics of the SEESCOA component-based software architecture. All additional functionality



(a) Single Port (b) Multi Port (c) Multicast Port

Fig. 1. SEESCOA Ports

must be added through extension modules that can be hooked onto the DRACO core. Figure 2 represents the high-level design of the DRACO system. The left part represents the core, the right part the optional modules and the circles at the left the shell(s) that give external access to the system.

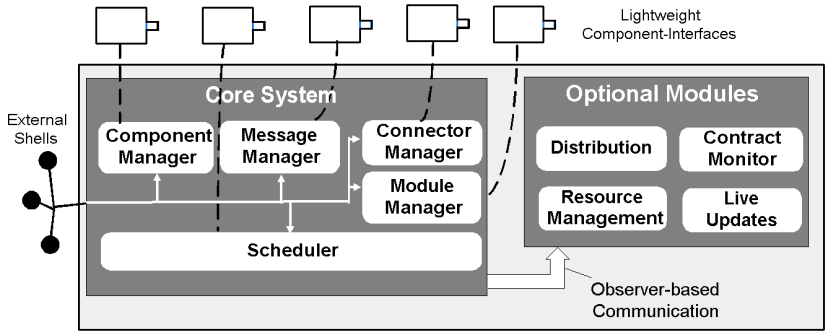


Fig. 2. High-level design of the DRACO runtime system

Each of the core modules implement a strict interface, which allows them to cooperate in a predefined manner. At startup, the core is dynamically constructed based on an XML configuration file describing which implementation to use for each of the core modules. The ability to customize the core modules makes DRACO an appropriate platform for various kinds of research (e.g. experiments with new scheduling algorithms by replacing the default scheduler). The core system, however, can not be dynamically reconfigured once instantiated. The DRACO core modules are:

Component Manager This module is responsible for instantiating components and their ports. It keeps track of them and allows looking them up.

Connector Manager This manager is responsible for creating and maintaining connectors between component ports. Each connector has associated

message handlers for guiding the message delivery. Handlers can be added by other modules to monitor or to alter the message flow.

Scheduler The scheduler is responsible for scheduling the messages for delivery. It ensures that the order of the messages is preserved for each port. It is also the task of the scheduler to make sure that no more than one execution thread accesses a component at any point in time.

Message Manager The message manager guides the messages sent by a specific port towards the connector that is associated with that port.

Module Manager This core module allows to add extension modules to the core system dynamically at execution time. These extension modules add any kind of functionality to the core. This property makes the system extremely flexible. Modules can access all core modules directly and can subscribe to each core module for numerous events, which allows them to interfere with the normal functional flow inside the core system.

The applications supported by this middleware are composed of interconnected components that send asynchronous messages to each other via their ports. When a message is sent, it first travels through a chain of send message handlers which, in the end, deliver the message at the scheduler where it is queued for execution. As soon as the scheduler is ready to process the message, it assigns one of its threads to forward the message through a chain of receive message handlers that deliver the message to the receiving port. After this, the message is processed by the component's implementation that matches the message type. Finally the thread is released to the scheduler again.

The optional modules can subscribe to numerous events at each of the core modules, e.g. when the scheduler starts to process a new message, when the connector manager establishes or removes a connector, when the component manager loads a new component, etc.. They can also add their own send and receive message handlers to connectors for interacting or monitoring the message flow between ports. In many ways optional modules can hook up to the system. This is essential for flexible functional and non-functional extensions.

DRACO components are created in .component files using a slightly altered Java syntax that are first compiled to normal Java source files. Secondly, the component's Java source files are compiled to Java class files and bundled into a *jar* file. The CCOM tool [26] can help us to develop components and to compose applications by connecting component instances. It was a prerequisite of the SEESCOA methodology that components are simple entities that are suitable for creating applications that fit on embedded platforms. Realized test applications, such as the camera surveillance case (see section 5), prove the feasibility of the SEESCOA approach.

3.3 The DRACO Distribution Module

Supporting distributed systems has always been one of the most important goals for designing middleware [3]. Although the DRACO core system on its own does

not support distributed applications, an extra module can be loaded to enable this important additional functionality.

The *Distribution* extension module (DM) adds distribution functionality to the core platform in a complete transparent way. Therefore it introduces the notion of *proxy components*, similar to the proxy pattern defined in [8]. These proxy components are very light-weight components that represent remote components. They offer the same ports and exactly the same semantic information. From a black-box point of view, a component and its proxy are exactly the same. However, the proxy is able to forward the messages it receives to the distribution module.

Figure 3 illustrates how the journey of a message between remote components looks like by using proxy components. When `Foo` sends a message to `Bar_proxy`, the message is automatically forwarded by the distribution module on *Host X* to the distribution module on *Host Y*. This distribution module then orders the `Foo_proxy` component to resend this message through the same port as it was originally sent, which finally delivers the message at component `Bar`. One of the consequences is that the message has been scheduled twice, once on both hosts.

The distribution module has several responsibilities. Firstly, it is responsible for setting up and tearing down connections between remote DRACO systems. In the design of the DM, a connection is an abstract concept that can be implemented by any kind of physical wired or wireless connection. Only some predefined interfaces have to be implemented to create a new connection type. Secondly, the distribution module is responsible for managing proxy components and generating them based on real components. No stubs or other design-time entities are required for making components communicate in a distributed way, which incorporates a considerable advantage over traditional approaches that need additional constructs (e.g. the stubs and skeletons used by Java RMI[3]). Furthermore, the DM is responsible for handling the connector setup whenever a local connector is created between the ports of any component and a proxy component.

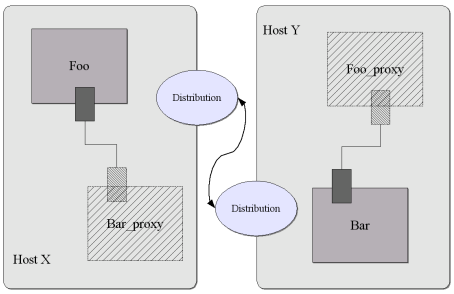


Fig. 3. Proxy Components

[3] Remote Method Invocation

From an external point of view, distributed communication in Draco is handled as follows. The distribution module is accessed through a Draco shell for setting up a new connection. Once the connection type and the connection parameters have been provided, the distribution module automatically loads the connection manager associated with the given connection type and initializes it. As soon as a connection is set up, the DM can be accessed through the shell to query the remote system for certain components and proxy components can be requested. Once a proxy component is available, a regular connector can be established between the port of the proxy component and a port of any other component.

When such connector is created, the distribution module, which is hooked up to the core system, gets notified and commands the remote distribution module to create a proxy component reflecting the local component whose port was connected with one of the proxy component's ports. This means that if in figure 3 only `Foo`, `Bar_proxy`[4] and `Bar` were available, `Foo_proxy` automatically gets created when a connector is established between `Foo`'s port and `Bar_proxy`'s port. The connector between `Foo_proxy` and `Bar` is then also automatically created.

An advantage of this approach is that it is also possible to request proxy components from existing proxy components, which allows for routing messages between components over several hosts transparently.

4 Providing Pervasive User Interaction

The characteristics of a pervasive computing environment also imply great challenges for providing the user with a consistent and transparent interaction mechanism for accessing the functionalities of particular services and compositions of services. These challenges originate from the immense diversity of possible user appliances and the distribution of components in the environment. Every device has its own characteristics: small/large/no screen, stylus/mouse/keyboard-/touch/speech interaction, etc.. It is clear that classical UI design techniques are not sufficient in such a heterogeneous and dynamic environment.

The main problem of these approaches is that the UI is hard-coded for each particular device. For this reason, porting an existing UI to a different device implies the redesign and the re-implementation of the complete UI to match the new device constraints. Because the UIs are hard-coded they also do not offer support for adaptability and migration. Adaptability is the property of a UI to adapt to changes in the environment while UI migration enables the transportation of UIs from one device to the other.

The aforementioned problems imply that a more general solution for user interaction handling is necessary. To enable this we introduce the use of Interaction Components (ICs). An IC is a user interface rendering component that serves as a router between users and components. ICs are typically located on one of the user's personal mobile devices where they provide a doorway for human users to the services of components that are available in the user's computing space.

[4] `Bar_proxy` could have been requested and initialized earlier by the user.

The Interaction Component ensures that the user interfaces of all components the user is interacting with are shown on the personal device. This interface is generated based on the high-level user interface descriptions (HLUID) provided by these components. In the following sections we elaborate on our choice for using high-level UI descriptions, the structure of the HLUID and the process of generating effective UIs from the HLUID used by the ICs.

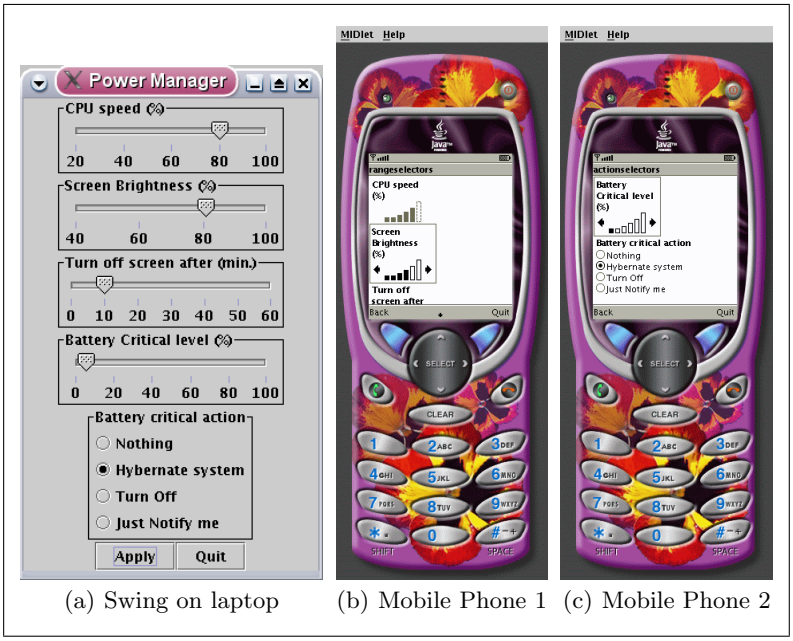
4.1 High-Level User Interface Descriptions

Based on the argumentation provided in [12] we use the eXtensible Markup Language (XML) [5] to describe UIs on a sufficiently high level. Many other initiatives confirm that using XML for describing UIs is a feasible choice [1, 6, 16]. Furthermore, when talking about the design of abstract models for describing UIs, important related work can be found from Thevenin and Coutaz [23] in which they introduce the plasticity property of UIs.

In our approach we attach XML-based HLUIDs to interactive Draco components running in the runtime. These descriptions provide information about the hierarchical structure of the interface, without relying on a particular kind of device. More specific, the UI is described in terms of Abstract Interaction Objects (AIOs) [27]. These AIOs are device and system independent and will be mapped on device and system dependent Concrete Interaction Object (CIOs) during a UI rendering process. The used mapping depends on the device on which the UI is instantiated (rendered). For example, a range is a type of AIO that can be mapped on a Scrollbar when using Java AWT, a JSlider when using Java Swing or a Gauge when using a Java MIDP enabled device. An extension of this approach toward multi-modal UIs is that AIOs can have a corresponding representation in speech. At this moment we have implemented support for Text-To-Speech where a text representation of a AIO is spoken as a response to an action of the user.

Although mapping AIOs to CIOs related to a particular device is not a straightforward process, this approach proves to be suitable [14]. It allows the interface to be transformed into a concrete interface taking advantage of the available input/output facilities of the device.

Figure 4 shows how one HLUID can be rendered for different devices. This happens without any manual intervention: the abstract interface is transformed automatically to a concrete interface which is suitable for the target device. This example describes a power manager one can use to configure the power settings of the device. When a laptop computer is available, the user sees all the possible settings on one screen (a). The concrete interface can change according to new constraints, e.g.: the small screen of the mobile phone implies that only a part of the available controls can be showed to the user (b+c). An abstract UI consists of one root group containing subgroups, which on their turn can contain other subgroups themselves (figure 5,[14]). At the leaves of this tree we find the particular AIOs that construct the UI. Some of these AIOs can trigger a particular operation upon the system when they are manipulated by the user. This characteristic is reflected by the presence of action elements in



(a) Swing on laptop (b) Mobile Phone 1 (c) Mobile Phone 2

Fig. 4. One description, many concrete User Interfaces

the HLUID (figure 5). For this work we extended the action element with the component port through which the message should be sent. In case the apply-button is pressed, a message *applyPowerConfig* is sent through the *Event* port of the IC that is responsible for this concrete UI. This means the IC that has rendered a UI will handle events triggered by the UI and distribute these to the application components it is working for. The parameters passed with the message are extracted from the *cpuspeed, screen, turnoffscreen, batterycritical* and *whatcritical* widgets that are defined in the HLUID (not shown in the figure).

4.2 Operation of the Interaction Component

As mentioned in the previous sections we use ICs which are a special type of Draco components. An IC typically resides on a user appliance and is part of a running Draco component system. The IC on the user appliance provides a presentation of the behavior of selected components to the user. Eventually the presentation will be adapted to the constraints of the particular type of device.

In our approach we defined two sorts of ICs. The first type is a general implementation that supports the generation of UIs using Java AWT, Java Swing, Java MIDP or HTML. The second one can be implemented for a particular device. This means every manufacturer of a device can implement a Draco IC for that device which will generate the UI in the device dependent widget set.

In section 4.1, we showed that each loaded interactive component can be annotated with a HLUID. When a user wants to interact with a particular

```
<?xml version="1.0" encoding="utf-8"?>
<ui>
  <title>Power Manager</title>
     .....
    <group name="submit">
      <interactor>
        <button name="applybutton">
          <info>Apply</info>
          <action>
            <func>applyPowerConfig</func>
            <param>cpuspeed</param>
            <param>screen</param>
            ........
            <port>Event</port>
          </action>
        </button>
      </interactor>
    </group>
  </group>
</ui>
```

Fig. 5. A piece of the abstract UI description for the interface in figure 4

component or composition of components, the IC on his device sends a *GetUiDescription* message to the destination. This message is sent through the *UiDescription* port (figure 7). The receiving component responds to this request by sending the HLUID to the IC which interprets the HLUID and renders an appropriate UI on the user appliance. Because we use XML for the HLUIDs they are easy exchangeable. Another advantage is that HLUIDs from different components can be composed very easy by the IC by attaching one XML tree to the other. The resulting HLUID is then rendered. Figure 6 shows the interface resulting from a merge process between the HLUID of a camera component and the HLUID of the motiondetector.

4.3 Adaptability to Device Constraints

Draco supports the deployment of new services at runtime; this results in dynamic systems where services appear and disappear while the system is running. Because of the dynamic nature of pervasive systems there is no certainty of the target devices that are used to interact with the service at design-time. We use the following mechanism to cope with rendering the UI for a heterogeneous environment: an IC can be deployed on a per-device basis. This means the IC knows the constraints of the device it has to use to render the HLUID it receives from other components. Besides custom mappings from AIOs to CIOs based on the available widget set on the device, an IC also has an automatic layout

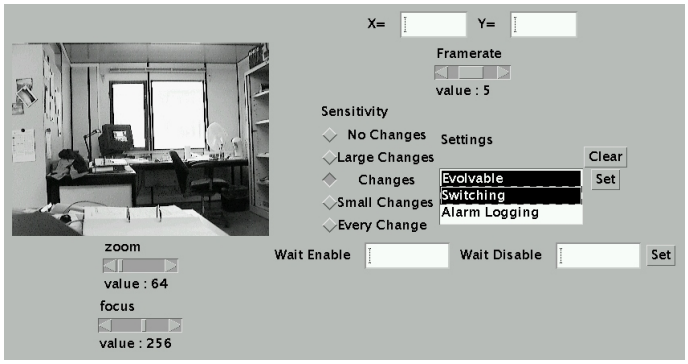


Fig. 6. Camera component UI (left) and motion detection component UI (right) merged into one interface.

manager to render the interface adapted to the screen constraints of the device. A simple constraint-based layout management algorithm is used for calculating an appropriate UI structure from a set of spatial-constraints [13]. Unlike most other layout managers, the one used in the ICs can automatically decide to split the UI into different coherent parts and present this to the user as a stack of cards that can be browsed (the different parts are put on top of each other). Figure 4 shows how all controls are shown when there is enough screen space, and only a part of the controls is shown when the screen space is limited. The other controls are also available through navigation.

It is also possible to distribute the UI over a set of devices. When a service or a group of services can be visualized on a set of different cooperating devices, no change is necessary in the infrastructure described in the previous sections. Each of the devices has its own IC, taking a part of the XML-document for rendering. Splitting up the complete XML-document into appropriate"sub"-documents is supported by the layout management algorithm. The biggest disadvantage of this kind of adaptability is the unpredictable usability of the concrete UI that will be rendered. For now, there is no support for the designer to enforce guidelines for ensuring that the UIs that will interact directly with the human user are usable.

5 Case Study

We evaluated the Draco platform and the Seescoa design methodology in a test case in which we have built a camera surveillance system. This camera surveillance system can be used for security related purposes such as physical intrusion detection and registration of activity in home and office buildings. An embedded device[5] connected to a digital camera[6] serves as our observation station. It is interconnected through a TCP/IP network to a desktop PC which

[5] We used a PC/104 system, see http://www.controlled.com/pc104/
[6] A DFW-VL500 firewire camera

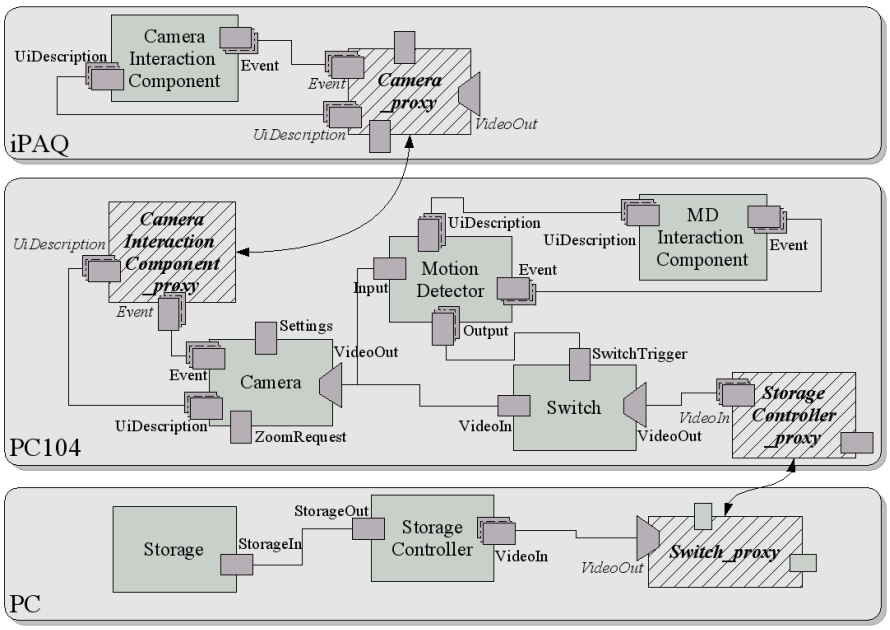


Fig. 7. Overview of the Camera Surveillance Case

is deployed as storage and control station. The embedded device has a processor working at 233 Mhz, 32 MB memory, a 16 MB flash disk holding the operating system, a Java virtual machine, our component system and the test case code.

Figure 7 gives an overview of the component compositions in the surveillance case. Each box indicates that the components are deployed on different devices. The central *Camera* component continuously grabs images from the camera at a predefined rate and multicasts them towards the *Motion Detector* and the *Switch* component. The motion detector analyzes the images and produces an *alarm-start* output message when motion is detected. The switch, receiving the message from the motion detector, forwards the video stream towards its output port until the *alarm-stop* message is received, meaning that the motion has ceased. The suspicious images are sent to the *Storage Controller* component, which is located on another host (the logging server). Proxy components are introduced for handling the remote communication transparently. The *Storage* component, which encapsulates database access, finally stores the images.

In addition to the core application, two graphical user interfaces are rendered onto the devices: the UI for the motion detector on the embedded device on a small LCD display and the one for the camera remotely on the logging server. Again, two proxy components were introduced for supporting the remote interaction between the *Camera Interaction Component* and the *Camera* component. An alternative user interface could have been generated as depicted in figure 6. In this figure, both the user interface of the camera and the motion detector

are combined and merged into one single interface. This is realized by simply connecting only one *Interaction Component* to both the *Motion Detector* and the *Camera* component.

6 Future Work

In the near future we would like to extend our approach to satisfy even more needs for the design of interactive systems that are deployable in AmI environments. This work will be embodied by the CoDAMoS project. CoDAMoS stands for Context-Driven Adaptation for Mobile Services. This project will elaborate on context-aware services and UIs, Quality of Service (QoS) awareness, service management techniques and code mobility. The definition of context awareness we will use covers several aspects such as user preferences, location awareness (the physical location) and device resources. Each of these aspects need advanced discovery, monitoring and querying techniques to realize them, but the advantage is a computing environment with powerful support for service cooperation and context-aware, adaptable and multi-modal UIs. A context aware infrastructure also enables QoS awareness both on device level (through device resources such as memory, bandwidth and CPU power) as on inter-service level (cooperation quality such as reliability, timing and correctness). Therefore, the Draco runtime environment will be thoroughly extended with application adaptation engines that can relocate, compose and decompose applications accordingly to the user's needs and the available resources. Furthermore, service management is needed to handle service deployment that controls on which device services are deployed based on QoS information.

7 Conclusions

In this paper we presented a component runtime infrastructure for pervasive environments, Draco, that enables the easy creation of distributed, interactive applications while providing a mechanism for consistent user interaction in pervasive environments. The Draco middleware supports the Seescoa component-oriented design methodology. This means that a clear distinction is made between the concepts components, ports, connectors and contracts.

The ccom tool provides the means for designers to help them in building distributed applications for Draco while the approach taken for managing user interaction enables the construction of device and platform independent UIs. This results in an integrated environment for building and deploying pervasive systems.

The case was developed as a proof-of-concept for the methodology and tools developed and introduced in this paper. Several software component developers cooperated to create a camera surveillance system; while there was no developer with any prior experience in UI design, annotating their components with abstract UI descriptions did not raise any problems. This proves the practical

relevance of our approach for attaching high-level user interface descriptions to components.

8 Acknowledgments

Part of the research at EDM is funded by EFRO (European Fund for Regional Development), the Flemish Government and the Flemish Interdisciplinary institute for Broadband technology (IBBT). The SEESCOA (Software Engineering for Embedded Systems using a Component-Oriented Approach) project IWT 980374 and CODAMOS (Context-Driven Adaptation of Mobile Services) project IWT 030320 are directly funded by the IWT (Flemish subsidy organization).

References

1. Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. *UIML: An Appliance-Independent XML User Interface Language*. World Wide Web, `http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html`, 1998.
2. The OSGI Alliance. Osgi. Internet. http://www.osgi.org.
3. David E. Bakken. *Encyclopedia of distributed computing*. Kluwer Academic Press, 2001.
4. ObjectWeb Consortium. Fractal component model. `http://fractal.objectweb.org/`.
5. World Wide Web consortium. *Extensible Markup Language (XML)*. World Wide Web, `http://www.w3.org/XML/`, 2001.
6. Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In *IUI 2001 International Conference on Intelligent User Interfaces*, pages 69–76, 2001.
7. Gordon S. Blair et al. The design of a resource-aware reflective middleware architecture. *LNCS in Meta-Level Architectures and Reflection (Reflection'99)*, pages 115–134, 1999.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
9. IST Advisory Group. Ambient intelligence: from vision to reality. ISTAG Reports. http://www.cordis.lu/ist/istag-reports.htm.
10. T.D. Hodes, R.H. Karz, E. Servan-Schreiber, and L.A. Rowe. Composable Ad-hoc Mobile Services for Universal Interaction. *Proceedings of The Third ACM/IEEE International Conference on Mobile Computing (MobiCom '97)*, pages 1–12, 1997.
11. Chris Johnson, editor. *Interactive Systems: Design, Specification, and Verification, 8th International Workshop, DSV-IS 2001, Glasgow, Scotland, UK, June 13-15, 2001, Revised Papers*, volume 2220 of *Lecture Notes in Computer Science*. Springer, 2001.
12. Kris Luyten and Karin Coninx. An XML-based runtime user interface description language for mobile computing devices. In Johnson [11], pages 17–29.
13. Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device Layout Management for Mobile Computing Devices. Technical Report TR-LUC-EDM-0301, Limburgs Universitair Centrum – Expertise Centre for Digital Media, September 2003. Available at `http://www.edm.luc.ac.be/english/research/publications/139.html`.

14. Kris Luyten, Chris Vandervelpen, and Karin Coninx. Migratable User Interface Descriptions in Component-Based Development. In Peter Forbrig, Quentin Limbourg, Bodo Urban, and Jean Vanderdonckt, editors, *Interactive Systems: Design, Specification, and Verification*, volume 2221 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2002.
15. Sun Microsystems. Jini network technology. `http://wwws.sun.com/software/jini/`.
16. Andreas Mülller, Peter Forbrig, and Clemens Cap. Model-Based User Interface Design Using Markup Concepts. In Johnson [11], pages 30–39.
17. Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. *Lecture Notes in Computer Science*, 2201:56–??, 2001.
18. Shankar R. Ponnekanti, Luis Alberto Robles, and Armando Fox. User Interfaces for Network Services: What, from Where, and How. *Fourth IEEE Workshop on Mobile Computing Systems and Applications* , pages 138–??, 2002.
19. Peter Rigole, Yolande Berbers, and Tom Holvoet. Bluetooth enabled interaction in a distributed camera surveillance system. In *To appear in proceedings of The Wireless Personal Area Networks Minitrack at HICSS 2004*, Big Island, Hawaii, January 2004.
20. Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, fifth edition edition, 1998.
21. Artima Software. The serviceui project. `http://www.artima.com/jini/serviceui/`.
22. Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998. ISBN 0-201-17888-5.
23. David Thevenin and Joëlle Coutaz. Adaptation and Plasticity of User Interfaces. In *Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users*, 1999.
24. David Urting, Stefan Van Baelen, and Yolande Berbers. Embedded software using components and contracts. In *Proceedings of ECOOP 2001, SIVOES workshop*, Budapest, Hungary, June 2001.
25. David Urting, Stefan Van Baelen, Tom Holvoet, and Yolande Berbers. Embedded software development: Components and contracts. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 685–690. ACTA Press, 2001.
26. David Urting, Yolande Berbers, Stefan Van Baelen, Tom Holvoet, Yves Vandewoude, and Peter Rigole. A tool for component based design of embedded software. In *Proceedings of TOOLS Pacific 2002*, Sydney, Australia, February 2002.
27. J. Vanderdonckt and F. Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *ACM Conference on Human Aspects in Computing Systems InterCHI'93*, pages 424–429. Addison Wesley, 1993.
28. Yves Vandewoude, David Urting, and Yolande Berbers. Supporting evolution in seescoa. In *Workshop on Transdiscipliary Software Engineering at Seventh World Conference on Integrated Design & Process Technology*, 2003.
29. Mark Weiser. The Computer for the 21st Century. In *Scientific American*, 1991.