Towards Dynamic and Cooperative Multi-device Personal Computing

Spyros Lalis[1], Anthony Savidis[2], Alexandros Karypidis[1], Jurg Gutknecht[3], and Constantine Stephanides[2]

[1] Computer and Communications Engineering Department, University of Thessaly, Thessaly, Greece
[2] Institute of Computer Science, Foundation for Research and Technology Hellas, Heraklion, Crete, Greece
[3] Computer Science Department, Swiss Federal Institute of Technology, Zürich, Switzerland

1 Introduction

The significant technological advances in hardware miniaturisation and data communications change the landscape of computing in a profound way. A rich variety of sensing, storage and processing nodes will soon be embedded in artefacts and clothes worn by people. Numerous computing elements will be integrated in appliances, furniture, buildings, public spaces and vehicles. It now becomes possible to move beyond the physical but also mental boundaries of the desktop, and to develop novel forms of computing that will efficiently support people in their daily activities without constantly being in the center of their attention.

In the 2WEAR project[1], we have explored the concept of a system that can be formed in an ad-hoc fashion by putting together several wearable, portable and infrastructure devices that communicate via short-range radio. This setting deviates from the desktop paradigm in significant ways. What we usually refer to as "the" computer becomes a collection of different autonomous elements that co-operate in a dynamic fashion without relying on a pre-arranged setup. Various applications and units of functionality reside on separate devices that can be widely heterogeneous in terms of computing and user interaction resources. Moreover, the number and type of elements comprising the personal system can change at any point in time. Using proximity as a natural form of control, the user may produce a wide range of system configurations, leading to different device interactions and application functionality. As a step towards this vision, we have worked towards advanced mechanisms for dealing with yet at the same time exploiting the heterogeneous and dynamic nature of such a system, with minimal programmer and user intervention.

In the following, we put our work in perspective of the research done in the area of ubiquitous computing. We continue by giving a motivating scenario and an overview of our system architecture. Then, we present in more detail the most important elements of the 2WEAR system: (i) an open and flexible communication framework; (ii) support for distributed user interfaces; and (iii) seamless distributed storage management. Finally, we revisit our scenario, indicating how it can be actually implemented using our system.

[1] http://2wear.ics.forth.gr

N. Streitz, A. Kameas, and I. Mavrommati (Eds.): The Disappearing Computer, LNCS 4500, pp. 182 – 204, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Background and Related Work

The vision of ubiquitous or invisible computing, formulated several years ago by Marc Weiser (Weiser 1991), has inspired numerous research efforts during the last years. Work covers a broad spectrum of technical and scientific areas, including wireless networks, special purpose devices, embedded sensors, distributed and mobile computing, runtime and middleware systems, activity recognition, application composition frameworks, human computer interaction, and security and privacy. It also crosscuts many application sectors, like the professional and business domain, classroom and home environments, mission critical computing and healthcare. In the following we put our work in perspective and briefly discuss related work[2].

A distinguishing characteristic of the 2WEAR project is its focus on the ad-hoc combination of several wearable and portable devices to form a single personal computing system. Unlike some major research efforts on mobile, ubiquitous and pervasive computing systems (Brumitt et al. 2001; Garlan et al. 2002; Johanson et al. 2002; Kunito et al. 2006; Roman and Cambell 2000), in our work there is no strong reliance on a smart infrastructure. By design, applications may reside on different devices, and devices providing auxiliary resources can be replaced by others to continue operation, with degraded or enhanced functionality. Infrastructure-based resources found in the environment can also be exploited in an opportunistic fashion.

Most work on mobile and wearable computing focuses on context acquisition and activity recognition, ergonomic design or investigates new forms of user interaction based on special sensors and input/output devices (Amft et al. 2004; Bang et al. 2003; Gellersen et al. 2002; Krause et al. 2003; Ogris et al. 2005; De Vaul et al. 2003). However, system configuration is typically fixed, based on a single wearable or portable computer, and several peripherals wired to it. 2WEAR takes a largely complementary approach, targeting a multi-device ad-hoc personal computing environment whose configuration may change at any point in time. The Spartan Bodynet (Fishkin et al. 2002) follows a similar approach, letting embedded applications interact with different wireless peripheral components, but there seems to be little advanced support in terms of handling a changing device configuration during application execution. The SoapBox system (Tuulari and Ylisaukko-Oja 2002) employs short-range radio for the special case of letting wearable sensors communicate in a flexible way with a central processing node, typically a personal computer or laptop. Pursuing a somewhat extreme model, the Personal Server system (Want et al. 2002) advocates a single portable device holding all data and applications, which can be accessed over wireless via PC-based terminals found in the environment using a web-based protocol. However, applications residing on the Personal Server cannot exploit the resources of other wearable and portable devices. The wearIT@work project (Kuladinithi et al. 2004) investigates wearable computing in conjunction with many wireless devices which may belong to different persons, but focus is mainly on mesh networking issues rather than higher-level forms of application support. Contrary to 2WEAR and the aforementioned systems, which use radio-based communication, there is also

[2] Even though we try to provide indicative references, we most certainly do not give a comprehensive survey of work on ubiquitous and wearable computing, which is well beyond the scope of this article.

significant work on intra-body communication and corresponding user interaction metaphors, such as "touch and play" (Park et al. 2006).

In terms of service / resource discovery and invocation, our work has common characteristics with other architectures designed for ad-hoc distributed computing systems, such as MEX (Lehikoinen et al. 1999) or MOCA (Beck et al. 1999). The main difference is that we do not commit to a specific programming language, middleware API or application model. Interoperability is achieved via an open and platform neutral communication protocol and encoding, which can be easily supported in different languages and runtime environments. Location and distribution (programming) transparency, if indeed desired, is achieved by introducing language- and runtime-specific libraries and/or proxy objects. Our approach is thus more in the spirit of web technologies, but we do not use HTTP or XML which introduce considerable overhead. Some middleware systems, such as PCOM (Becker et al. 2004), provide considerable support for dynamic service (component) selection and binding at runtime. Our system provides similar functionality, but this is specifically tuned for the purpose of user interface management.

The user interface framework developed in 2WEAR, named Voyager, offers to the application programmer a library of abstract interactive components that utilize distributed user interface services hosted by proximate wearable, portable and ambient devices. Dynamic service discovery, negotiation, synthesis and adaptation occurs in a transparent fashion to application programmers, hiding all underlying user interface management details, while ensuring state persistence and automatic dialogue recovery upon disconnection or device failure. Related work in the area of ubiquitous and wearable computing is mostly concerned with the design and implementation of ambient dialogues, context awareness and adaptive interfaces. For instance, the work reported in (Banget al. 2003) investigates new types of interaction devices suited for traditional static wearable interactions. The notion of context awareness is related to ways of sensing information from the physical and computational environment (Abowd and Mynatt 2000) making a system aware of the context in which it is engaged. However, there are no propositions on how to address the situation where the context can be the interface as well; like in Voyager. The need to effectively reflect such awareness in interaction is discussed under the concept of plasticity in (Calvary et al. 2001), emphasizing the requirement for distributed applications to "withstand variations of context of use while preserving usability", but no full-scale development framework is proposed. There has been work regarding the engineering of context-aware applications based on re-usable higher-level software elements, called widgets (Salber et al. 1999). These are responsible to notify the run-time application of various types of events occurring in the environment, such as particular activities, or the presence of persons. The Voyager framework adopts a similar toolkit-based approach in order to support distributed and adaptive user interfaces. Dynamic adaptation is also provided in the SUPPLE system (Gajos et al. 2005), but in this case focus is on the rendering / placement of graphical user interface components on a single display as a function of its size and user input statistics.

Storage management in 2WEAR is based on a peer-to-peer model where individual wearable and portable storage devices are viewed as autonomously collaborating and self-organizing elements. This makes it possible to perform several file management tasks such as backup, offloading and replication in an automated and asynchronous

fashion, requiring little or no explicit user input. Traditional network file systems like Coda (Satyanarayanan 2002) focus on more conventional client-server interactions. They support mobile clients and disconnected operation, but assume a single device per user, typically a laptop. Other ad-hoc file systems (Yasuda and Hagino 2001) allow users to flexibly share their data without relying on any infrastructure. The difference with our approach is that we do not support direct file access over the network. Programs wishing to read or write a remote file must explicitly create a local copy. Autonomous data exchange between portable devices is supported in the Proem system (Kortuem et al. 2001), by plugging into the system application-specific tasks, called peerlets. The main difference is that we introduce the basic file transfer functionality at the system level, via asynchronous operations to tolerate slow and intermittent connectivity.

3 Motivating Scenario and System Overview

We begin by giving an example of the envisioned system in operation: *"Mary arrives in City-X. At the airport she is given a city-guide application card, which she activates and puts in her pocket. She takes a bus downtown to explore the old city center. The application tracks her position using a GPS integrated in her backpack. Each time she is close to a site of general interest, a message appears on her wristwatch. At the click of a button, more information about the site is shown on the display of her camera, if turned on. She takes pictures using her camera, which are annotated with the GPS coordinates. When the camera starts running out of space, old images are offloaded on her wallet. Mary bumps into Jane who also happens to be visiting the city. They sit at a café and Mary takes out her camera to show Jane the pictures she took during the day. Meanwhile, the photos are being backed up at her home computer via a nearby access point. Jane expresses her interest in several of them. Mary marks them for access by Jane who switches on her data wallet. Feeling a bit hungry they decide to have a bite at a cozy restaurant suggested by the city-guide. Mary pinpoints its location on a map, using the café's table-display to view a large part of the map in detail. As they walk down the road, Jane's wallet silently copies the selected pictures from Mary's camera."* This short scenario already illustrates the potential wealth of interactions between devices that are carried by persons or encountered in the surrounding environment. It also suggests that such a computing system exhibits several key properties, briefly listed below:

The system is inherently distributed and heterogeneous. Different functions and resources are provided by various devices and artifacts. For instance, the city-guide application uses resources provided by other devices such as the GPS, the wristwatch and the table-display of the café, and the camera uses the GPS to annotate pictures and the wallet to store them.

The system is extensible and adaptive. New devices can be dynamically added or removed, and applications adapt their behavior as a function of resource availability. For example, the city-guide can be controlled via the wristwatch or the camera controls, and can display the map on the camera or table-display. Also, applications exploit the storage capacity of nearby devices and file backup is performed when close to an access point.

The system is largely autonomous. Most system-level interactions remain invisible and adaptation seldom requires explicit input. For example, the city-guide application automatically adapts to changing user interface resource availability (wristwatch vs camera resources), and data transfers between the camera, Mary's wallet, Mary's home computer and Jane's wallet are performed behind the scenes.

In pursuit of this vision, we have implemented our system according to the architecture, shown in Figure 1. At the low level, basic communication and discovery functionality is provided so that programs may access the resources that are available at any point in time; contributed to the system by the nearby devices and infrastructure. On top of it, more advanced functionality is provided in the form of a "vertical" exploitation of particular resources to relieve the application from having to deal with the details of a (changing) underlying system configuration. We have investigated two separate aspects which we felt were of particular importance, namely distributed user interface and storage management, which are actively supported via corresponding mechanisms. No attempt was made to implement this functionality in the form of a common middleware or API. Instead, project partners were free to choose their own level and flavor of support, depending on the characteristics of their development platform.

Our system prototype includes custom hardware (Majoe 2003), such as an embedded GPS, a wristwatch with a small display and four buttons, and a small wearable computer with no user interface elements (see Figure 2). The latter is used as a generic data and computation "brick", which may host a variety of subsystems and applications;

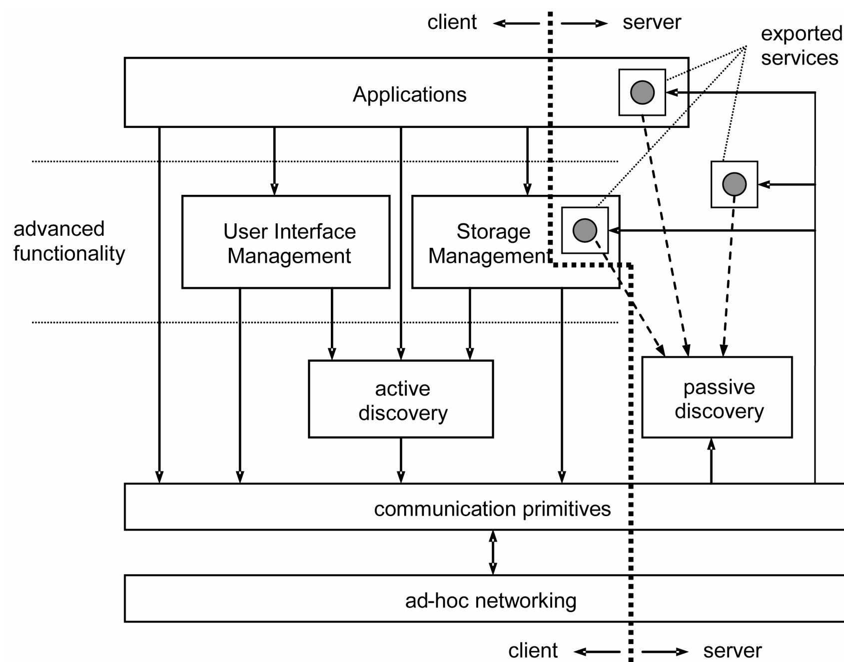

Fig. 1. Overview of the 2WEAR system architecture: arrows show the main control flow (invocation direction) between the various subsystems; the dotted line separates between client- and server-related functionality

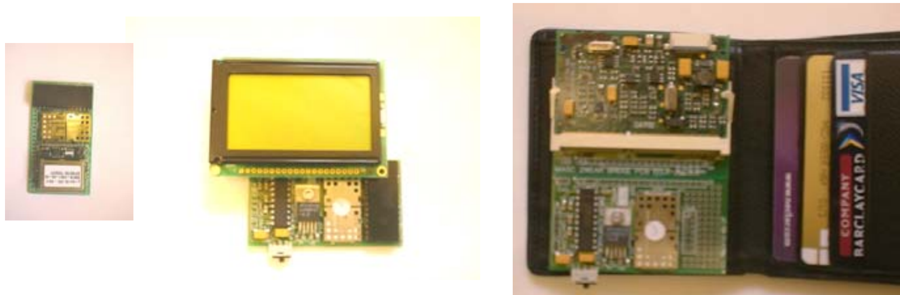

Fig. 2. Custom-made prototype devices

a personal system may include several such bricks, e.g. with different functionality assigned to each one of them. PDAs and laptops are also used to emulate other devices, such as a digital camera or a large public display and keyboard. Ad-hoc communication is implemented using Bluetooth radio (Bhagwat 2001)[3]

We note that the runtime setup is not necessarily the same for all devices. Special-purpose peripheral devices, such as the GPS and the wristwatch, have a minimal discovery and communication functionality installed in order to make their resources available to the system. On the other hand, the wearable computer (brick) or a PDA may be equipped with the full-fledged 2WEAR functionality to support local applications. Between these extremes, a device may only feature the storage management or user interface management subsystem, depending on its role and application requirements.

4 Open and Flexible Communication Framework

The envisioned system includes many heterogeneous devices which interact with each other in an ad-hoc fashion. Thus a key problem is to implement interoperable discovery and access, while catering for the significant differences in the computing resources of the various embedded, portable and infrastructure-based platforms. This is achieved via an open and flexible communication framework, described in the following.

4.1 Remotely Accessible Resources as Services

Interaction between devices follows a service-oriented approach. The concept of a service is used to denote a distinct resource or functionality that is accessible to remote clients over the (wireless) network. Any device may *export* a service to its environment and, conversely, *invoke* services being provided by other devices. The notion of a service in no way predetermines its implementation, and in this sense is orthogonal to the notion of a hardware or software component. Applications can, but do not have to, use services exported by devices, and may also provide services themselves.

[3] Bluetooth was chosen primarily because (at that time) it was the only mature short-range ad-hoc radio technology, with rudimentary support for most operating systems, including Windows and Linux.

Services are categorized using an open type system. Each service type is identified via a unique name, associated to: (i) a set of attributes; (ii) an access protocol; and (iii) a description document, which gives the semantics of the service attributes and access protocol. Attributes have as values ASCII strings. If it is desired to constrain an attribute value, the allowed expressions must be specified in the corresponding service description document. Service providers must supply a value for each attribute, i.e. specify a concrete service descriptor, and adhere to the respective access protocol. This concept is illustrated in Figure 3.

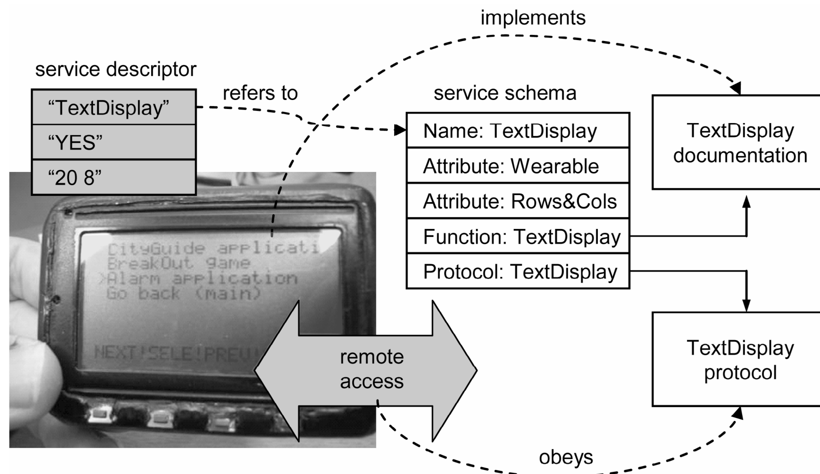

Fig. 3. A wristwatch device providing a TextDisplay service

4.2 Syntax-Based Protocols

Service access protocols are defined as *formal languages* that govern the data exchange between an entity invoking a service (client) and an entity implementing it (provider). The corresponding grammar is given using the Extended Backus-Naur Formalism (EBNF) (Wirth 1977) with some modifications.

Each protocol is specified as a set of productions that take the form <name> "=" <expression> ".", where <name> is the label of the production, and <expression> is an expression built out of labels, operators and tokens. There are five operators: (1) concatenation, denoted by a space character; (2) alternative, denoted by "|"; (3) grouping, denoted by "(" ")"; (4) option, denoted by "[" "]"; and (5) repetition, denoted by "{" "}". Labels occurring in expressions refer to productions whereas tokens refer to elements of the underlying alphabet[4]. To capture the bidirectional nature of communication, we augment the symbols appearing in expressions by a *direction* mode, denoted by normal and underlined text, indicating whether tokens travel from the client towards the provider or vice versa.

---

[4] Although tokens are the *atoms* that any production can finally be resolved into, their values need not be single bytes or characters; see token types.

The basic set of token types is listed in Table 1. Constants are given in the form of literals with an optional type cast to alleviate ambiguities. As we found that the extensive use of constants severely compromises the readability of protocol definitions, we introduce special symbols referred to as *keywords*. These are defined using an enumeration-like production, and are (per definition) mapped to consecutive natural numbers starting from zero. Since tokens are issued over a network, a corresponding type-safe serial encoding was also specified.

Table 1. The token types of the 2WEAR protocol syntax

Token name	*Description*
CHAR	ASCII character
STRING	Sequence of ASCII characters (zero terminated)
INTEGER	Integer (variable value-based length encoding)
BOOLEAN	Boolean
UUID	Universal unique identifier
BBLOCK	Byte block (length is specified as a prefix)
KEYWORD	Symbolic names

Listing 1 gives a few indicative examples. The TextDisplay protocol defines a transaction for writing an ASCII character at a given column and row, and receiving a failure/success feedback. The Alarm protocol describes a notification transaction, for setting a threshold and then continuously receiving (several) values. The RequestReply protocol defines a generic exchange of STRING tokens whose contents are left "open", thus can be used as a tunnel for higher-level scripting protocols. The Teller protocol specifies a simple e-banking dialog.

```
TextDisplay = {WriteChar}.
WriteChar = CHAR Column Row BOOLEAN.
Column = INTEGER.
Row = INTEGER.

Alarm = Threshold {Value}.
Threshold = INTEGER.
Value = INTEGER.

RequestReply = STRING STRING.

Teller = Login Password [OK {Action} | NOK].
Action = Balance | Deposit | Withdraw.
Balance = BALANCE INTEGER.
Deposit = DEPOSIT INTEGER [OK | NOK].
Withdraw = WITHDRAW INTEGER [OK | NOK].
Login = UUID.
Password = INTEGER.
Keywords = BALANCE DEPOSIT WITHDRAW OK NOK.
```

Listing 1. Indicative syntax-based protocol definitions

This formal approach enables a protocol specification to be used as a syntactic contract between developers who implement service clients and providers in a decoupled fashion. In addition, a wide range of interaction schemes, which go well beyond a simple request-reply pattern, can be expressed in a straightforward manner. Note however that the service-related semantics associated with a protocol cannot be inferred from its syntax and must be specified in the corresponding description document.

Protocol definitions are independent from programming languages and runtime systems, so that developers are free to design and implement the primitives that are most suitable for their system. Indeed, the flavour and level of application programming support in 2WEAR varied significantly, ranging from simple templates used to produce efficient monolithic code for embedded devices, to libraries providing general-purpose communication facilities (and protocol parsers) for different operating systems and languages. The minimal requirement for the transport layer on top of which such communication mechanisms can be developed is the ability to exchange bytes in a semi-duplex and reliable fashion. In our prototype implementations, Bluetooth L2CAP was used as the underlying transport mechanism[5].

4.3 Service Discovery

We advocate a multi-device personal computing system whose configuration can be changed at *any point in time* and in an *out-of-the-loop* fashion, i.e. without devices being explicitly notified about it by the user. Programs wishing to exploit the services provided by remote devices must therefore be able to detect them at runtime.

Service discovery is implemented as a two-step procedure. In the first step, the wireless ad-hoc network is searched for new devices. This is done by employing the native Bluetooth discovery (inquiry) mechanism. When a device is detected, in a second step, it is queried about the services it provides via the so-called Home protocol, which allows a client to specify the desired service types and / or attribute value(s) in order to receive the respective contact (port) information. The Home protocol is quite similar to other client-driven discovery protocols, like Salutation (Pascoe 2001). The main difference is that it is defined as a proper syntax-based protocol, hence can be invoked using the standard 2WEAR communication facilities.

Just like any service-level access, the Home protocol requires the establishment of a (L2CAP) transport connection, which consumes energy and network resources. This is done in vain if the target device does not provide any service of interest to the querying party. As an optimization, we exploit the class-of-device (CoD) field of Bluetooth inquiry packets to encode service provision hints. This makes it possible to infer which types of services *are not* provided by a discovered device, without employing the Home protocol; which seemed to considerably enhance service discovery as well as the overall network connectivity, especially in the presence of many devices. Finally, device and service discovery information is cached locally to eliminate frequent querying over the network. Entries have an expiration time which is renewed whenever the remote device shows signs of existence. It is nevertheless possible for a

[5] We also developed a TCP/IP "adapter" in order to perform initial interoperability tests between the programs of different partners over the Internet, before deploying code on the various portable and wearable devices.

program to receive outdated information. This can be explicitly invalidated upon an unsuccessful attempt to contact the service.

5 Dynamically Distributed User Interfaces

The ad-hoc nature of the envisioned system leads to an unusually dynamic and heterogeneous environment in terms of user interaction resources. In general, as illustrated in Figure 4, it is possible to differentiate among two layers of interaction-capable devices: inner layer wearable devices, which the user may or may not carry, depending on the situation; outer layer environment devices that fall inside the communication range of the user's portable / wearable computing system. Wearable devices are typically owned by the user and do not vary significantly while on-the-move. On the contrary, environment devices are part of the physical infrastructure, thus their availability is expected to vary as the user changes location.

Ideally, applications should be constructed in a way that facilitates the dynamic exploitation of the proximate interaction-specific devices which are available at any point in time, without requiring extensive programming effort. Research and development work towards accomplishing this goal has resulted in the delivery of the *Voyager* framework, which provides considerable development support for dynamically distributed user interfaces, built on top of the discussed communication framework.

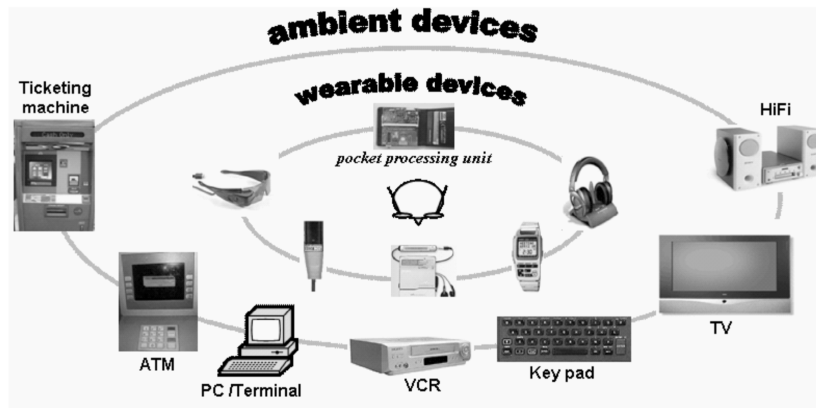

Fig. 4. Dynamically engaged ambient computing resources in mobile interactions

5.1 The UI Service Protocols

Due to the large diversity of user interface resources that can be provided by the various devices, it was mandatory to prescribe a common protocol that would be applicable to all possible UI service classes. Additionally, the protocol should allow programmers to detect and exploit on-the-fly any particular non-standard features offered by discovered devices. To this end, interaction with UI services occurs via three basic protocols: (i) the control protocol, (ii) the input protocol, and (iii) the output protocol. Slightly simplified definitions of these protocols are given in Listing 2.

```
Control = {GetP} [ACQ [OK {Main} RLS | NOK].
Main = GetP | SetP.
GetP = GETP PropId PropVal.
SetP = SETP PropId PropVal BOOLEAN.
PropId = STRING. /* scripting */
PropVal = STRING. /* scripting */
Keywords = ACQ RLS GETP SETP ACK OK NOK.

Input = INPUT {InEventId InEventVal}.
InEventId = STRING. /* scripting */
InEventVal = STRING. /* scripting */
Keywords = INPUT.

Output = OUTPUT {OutEventId OutEventVal BOOLEAN}.
OutEventId = STRING. /* scripting */
OutEventVal = STRING. /* scripting */
Keywords = OUTPUT.
```

Listing. 2. The basic UI protocols

The control protocol describes the communication for acquiring, releasing and checking the availability of a UI resource. It also allows clients to determine, query and modify the so-called properties of a resource. The input and output protocols dictate the reception of user input events from a UI resource and the issuing of user output events towards a UI resource, respectively. Every distinct UI resource is expected to support the control protocol, as well as the input and/or output protocol, depending on its type.

It is important to note that property / event identifiers and values are defined as STRING tokens, hence are open to definition and interpretation by service providers and application programmers. This makes it possible to accommodate extended introspection and parameterization functionality as well as to exploit the special UI capabilities of devices in a flexible way, without breaking the basic UI protocols. Of course, this introduces an additional level (or dimension) of service meta-information that must be appropriately documented by service providers and consulted by application developers. Some sort of coordination is also needed to ensure that identical STRING identifiers of properties and event classes actually imply identical semantics and compatible corresponding content-value expressions across all UI service implementations.

5.2 Primitive UI Services

Several UI resources, exported as corresponding UI services, were developed to provide rudimentary distributed input / output functionality: Button (input); Keyboard (input); TextDisplay (output); TextLine (output); GraphicsDisplay (output); Menu (input and output); TextEditor (input and output). Their functionality is summarized in Table 2. The access protocol for each UI service is defined by refining the basic UI protocols in a suitable way. In addition, to simplify programming, appropriate client-side APIs were implemented that hide the underlying discovery and communication process. For certain UI services several alternative implementations were provided on

different wearable and portable platforms, while others had to be emulated[6] on laptops and PCs.

Most of the UI service abstractions correspond to primitive input or output functionality, which are likely to be found in small wearable and embedded devices. However, services such as the Menu and the TextEditor encapsulate both input and output functions in a single unit. This reflects the situation where a device may provide advanced interactive behavior as a built-in feature. For instance, home appliances such as TVs employ menu-based dialogues, using their screen for output and their remote control unit for input. Thus it would be possible to export this UI functionality in the form of a distinct UI service, such as the Menu, which can be discovered and exploited by remote applications.

Table 2. The UI services

UI service type	*UI mode*	*Functionality*
Button	Input	receive button events
Keyboard	Input	receive key press events with key code
TextDisplay	Output	display a character at a given column & row
TextLine	Output	display a character at a given position
GraphicsDisplay	Output	display a bitmap
Menu	Input/Output	display list of textual options and receive as a result the choice of the user as a number
TextEditor	Input/Output	display text for (offline) editing and receive as a result a new text

5.3 Higher Level UI Management Support

Primitive UI services provide the basic functionality which can be used to implement a wide range of user interfaces. However, the application programmer is responsible for discovering the available services and combining them together to form meaningful interaction styles. Also, failure to communicate with a remote UI service must be handled, e.g. by replacing it with another available service of equivalent or similar functionality. Due to the inherently dynamic nature of the system, this becomes a commonly expected case rather than an exceptional situation, reflecting the need to accommodate dynamic UI re-configuration and deliver a persistent interaction throughout application execution.

To effectively provide such functionality at the application level, high-level *dialogue abstractions* were developed, taking into account the characteristics of small devices with restricted UI capability and relatively slow wireless communication. The adopted technique is based on virtual interaction objects supporting polymorphic platform bindings (Savidis 2005). For the purpose of the 2WEAR project, two generic dialogue object classes were introduced: (i) the Selector, allowing the user to select from an explicit list of displayed textual options; (ii) the TextEntry, enabling the user to enter / edit textual input. The application may employ an arbitrary number of such instances, but only one dialogue object instance may own the focus of user interaction at any point in time (i.e. a single focus object policy).

---

[6] All primitive UI services were emulated on a desktop environment to perform initial tests of the framework without relying on wearable / portable hardware.

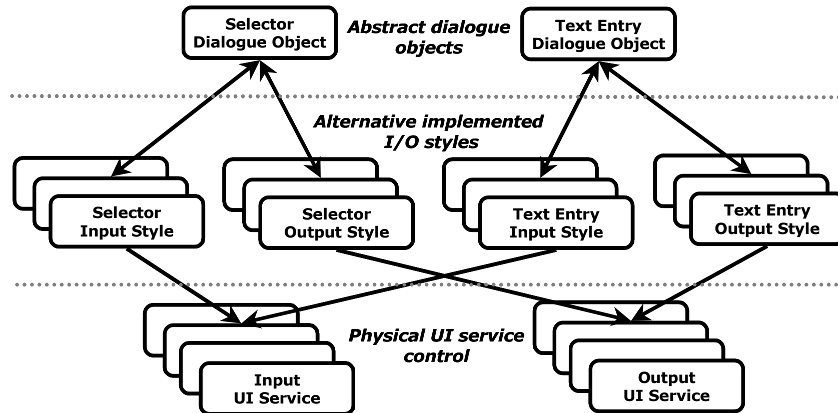

Fig. 5. Implementation structure for abstract (distributed) dialogue objects

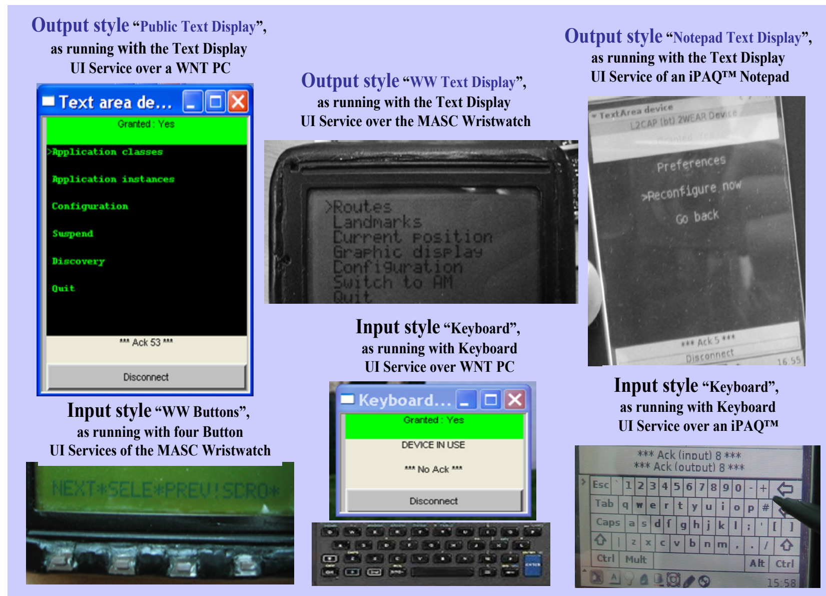

Fig. 6. Three alternative output (top) and input (bottom) styles for the Selector, based on different implementations of primitive UI services on various devices

These object classes encapsulate appropriate runtime control logic for the dynamic handling of UI services to achieve dynamic UI reconfiguration. This is accomplished through the implementation of alternative dialogue input and output control policies, also referred to as instantiation styles, as illustrated in Figure 5. Each style relies on a specific combination of UI services, and encompasses the code for managing its basic input / output functionality. Different input and output styles can be orthogonally combined to produce a large number of plausible physical instantiations for a given dialogue

object class. As an example, Figure 6 shows various styles that have been implemented for the Selector dialogue.

At runtime, the application may interact with the user as long as there are adequate UI services to support at least one input and one output style for the focus dialogue object. Else, the dialogue enters a stalled state, where it is attempted to discover new UI services in order to make at least one input and output style of the focus dialogue viable. In this case, the dialog reverts back to working state and user interaction is resumed.

UI service discovery is also possible when the application is in working state. This may enable additional I/O styles for the focus dialogue object, thus creating the opportunity to reconfigure a dialogue *in the midst* of user interaction. In order to control system behavior, the user can specify whether such on-the-fly optimization is desirable and introduce priorities between dialog instantiation options (different preferences can be given for each application). Table 3 summarizes the UI reconfiguration logic.

Table 3. Outline of the algorithms for UI re-configuration behavior, upon UI service discovery or loss, depending on the current dialogue state

	working	*stalled*
UI service discovery	(optimization round) if UI optimization is enabled and a preferable instantiation became viable, deactivate the current instantiation and activate the new one	(revival round) if one or more instantiations became viable, select the most preferred one, set the dialogue state to running and activate the new instantiation
UI service loss	if the lost service is used by the current instantiation, deactivate the current instantiation, set the dialogue state to stalled and perform a revival round	do nothing (it is impossible that an instantiation became viable with less resources than before)

The state of user interaction is centrally kept within the application's dialogue object instances. It is updated each time an input event is received from a remote UI resource, associated to its current input style, before communicating the effects back to the user via the respective the UI output resource of its current output style. More specifically, the Selector object records the option the user has focused on, while the TextEntry object records the text entered by the user and the current cursor position. This separation between remote input / output and application-resident processing with support for state maintenance is illustrated in Figure 7.

Since the dialogue objects and styles reside within the application, this state is preserved locally even if the communication with the dynamically employed (remote) UI services fails. As a consequence, when a dialogue object instance is resumed / reconfigured using a new input / output style and / or new corresponding UI services, these can be properly initialized to reflect the last state of the dialogue.

A special application, called the Application Manager, is used to initiate, terminate and give focus to other applications. Even though several applications may be running concurrently to each other, only one may own the application focus (i.e. a single focus

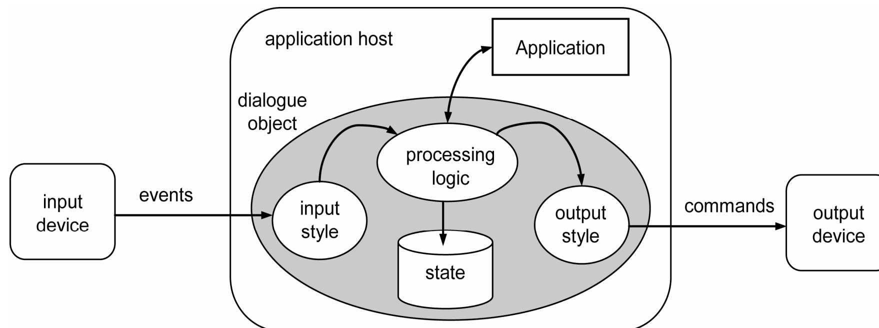

Fig. 7. Input / output processing of abstract dialogue objects; arrows indicate typical

application policy). Newly started applications are initialized in a suspended state, where no attempt is made to discover or allocate UI resources on their behalf. This is done only when an application receives focus, in which case a revival round is performed for its focus dialog object (see Table 2). The application then enters a working or stalled state, depending on the current availability of UI services. Conversely, when an application gives away the focus, it falls back to the suspended state, and the UI services allocated to its dialogue objects are released so that they can be used for the application that will receive the focus next.

6 Seamless Distributed Data Management

Data management becomes a key issue in a personal system that comprises many wearable and portable devices. Already today, the data that can be generated on the move, at the click of a button, can grow to exorbitant numbers. This imminent data explosion along with the fact that the number of personal devices used to generate, store and access data is most likely to increase in the future, aggravates the problem of data management. It quickly becomes clear that the conventional and *already* awkward model of moving data between devices via explicit and synchronous user commands does not scale. Even less so for embedded and mobile computing environments where devices have a limited user interface and user attention should be preserved.

With this motivation we developed a file-based[7] data management facility which turns storage devices into proactive self-organizing elements. This approach makes it possible to support several tasks in a largely automated way, requiring little or no explicit user input. For instance, we let wearable and portable devices collaborate with each other in an ad-hoc manner to copy or exchange files. Also, files generated on portable devices are forwarded via the Internet to a reliable storage service, referred to as the repository.

[7] This work focuses on (data) files generated by the user, rather than binaries and internal system and application-specific configuration files. We also assume, even though this is not enforced by our implementation, that files are immutable (Schroeder et al. 1985), i.e. files with the same name are guaranteed to have identical contents.

6.1 Architecture and Implementation Overview

The storage facility is implemented as a set of components which reside on personal and infrastructure devices and interact with each other to achieve the desired functionality. Figure 8 depicts a personal area network with two personal devices, a camera and a wearable brick (as a storage wallet), close to an access point through which the repository can be reached.

Portable and wearable storage devices run the *storage daemon*, a process that is responsible for performing the necessary interactions with other devices as well as with the repository. Applications running on portables use the *storage library* API to invoke the storage daemon[8]. The functionality of the repository is implemented via the *repository daemon* that handles requests received from remote storage daemons.

Finally, the *gateway daemon* acts as a network and protocol gateway, allowing portable devices to communicate with services that reside in the Internet, such as the storage daemon. The gateway daemon can be installed on any device (e.g. our wearable computer equipped with a GPRS modem) or infrastructure element (e.g. a PC plugged on an Ethernet network) that features a network interface through which Internet connectivity can be provided. The communication between the storage, repository and Internet access daemons is implemented using the basic 2WEAR communication mechanisms.

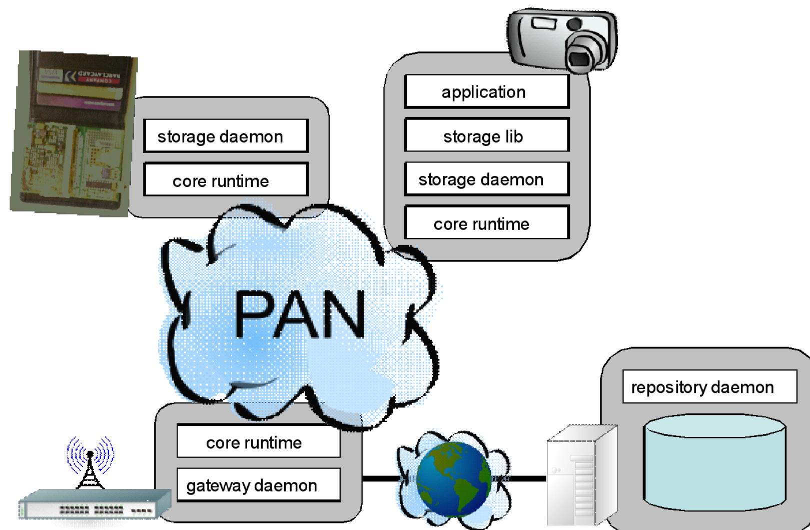

Fig. 8. The storage management architecture

6.2 Archival

The backup process is performed asynchronously behind the scenes with minimal effort on behalf of the application programmer. The storage library provides a *backup*

8 Some devices may act as “pure” storage data carriers, in which case they feature the device daemon without the device library or any local application.

operation to initiate backup for a given file. The backup status of a file can then be examined via the *isbackedup* operation.

The storage daemon maintains internally a queue of references to local files that need to be transferred to the repository. As long as this list is not empty, it periodically attempts to discover a gateway through which it can contact the repository daemon. When such a connection is eventually established, the storage daemon commences data transfer until all files have been successfully backed up at the repository.

Each entry of the backup queue is processed as follows. First, the file identifier (name) and size is sent to the repository daemon, which records this information and replies with a list of offset-length pairs indicating the missing file fragments. The storage daemon proceeds by sending the requested parts to the repository daemon. This is repeated until the repository replies with an empty fragment list, indicating that the entire file is safely stored, in which case backup proceeds with the next entry.

The transfer protocol is designed to support incremental data transfer on top intermittent connectivity, which is typical for mobile and ad-hoc computing environments[9]. As a result, if the connection with the repository breaks and is later reestablished, backup continues from the last fragment that was successfully received by the repository. Notably, an interrupted backup process can be resumed via another device holding a copy of the same file. If several devices concurrently attempt to backup the same file, the repository daemon will request different file fragments from each one of them, thereby achieving parallel backup.

6.3 File Collection, Offloading, and Replication

The storage daemon automatically creates free space on devices when storage is filling up. Reclamation is driven by two parameters[10]: the minimum free ratio (MFR) and the desired free ratio (DFR). When free space drops below the MFR, the device daemon attempts to remove local files. Candidates for deletion are the least recently accessed files that have been successfully backed up in the repository. Garbage collection stops when the DFR is reached.

It is however possible that the DFR cannot be met given the above restrictions. As a last resort option, the storage daemon attempts to offload a file on another device, which will assume responsibility for backing it up to the repository. When such a device is found, data transfer between the two storage daemons occurs employing a protocol that is almost identical to the backup protocol; making it possible to tolerate frequent disconnections. Notably, the receiving daemon is free to collect file fragments of an interrupted transfer at any point in time. The sending daemon may remove the file only if it was successfully received by another device.

As a special case of offloading, the storage daemon may copy a file on another device without removing it locally. Replication is meaningful if it is desirable to increase the availability of a file within the personal area network. Applications can specify the number of copies that should be created on different devices via the *setreplicas* operation.

---

[9] This is even more likely to occur in our system, since communication is implemented using a short-range radio technology.

[10] We assume these values as given. In practice, these could be set using a configuration tool or adjusted automatically using a meaningful system policy.

6.4 PAN-Wide File Access and Dissemination

The storage facility does not automatically provide a unified and up-to-date file system view for all neighboring devices. To implement this, it would have been required to frequently exchange file information between co-located storage daemons, consuming a significant amount of CPU time, bandwidth and energy. Instead, we decided to provide basic primitives that can be combined under application control to achieve similar functionality.

Local files that are to be made visible to applications residing on remote devices need to be specified in an explicit way. This is done using the *setexport* operation, which marks a file so that its existence can be determined by remote storage daemons. Export entries can be given a lifetime and are removed by the storage daemon upon expiration. If no lifetime is specified, the export status of a file must be revoked via *setexport*. To declare its interest in remote files, a program registers with the storage library a lookup expression and notification handler. As a result, the storage daemon starts to periodically query nearby devices for exported files that match this expression. In case of a match, the corresponding programs are notified by invoking their registered handler routine.

Programs access local files via the standard *open*, *read*, *write* and *close* operations. Direct access is not supported for remote files (that have been discovered via the via the lookup mechanism). If a program wishes to access a remote file, it must create a local copy via the *fetch* operation of the storage library. The corresponding data transfer over the network is performed in the background by the storage daemon, following the same protocol as for file replication, in the reverse direction. If the connection breaks, data transfer may be resumed, also by redirecting the request to another device that has a copy. Fetch operations are given an allowable silence period, indicating the maximum amount of time for establishing contact with a device that can provide the target file. If the storage daemon fails to commence or resume transfer within this period, it aborts the fetch operation and deletes the corresponding file fragments.

Listing 3 gives sample code for fetching all .jpg images found on nearby devices with a silence threshold of 5 minutes. The same mechanism is used to disseminate files of public / common interest to other people, e.g. family, friends and colleagues. The difference is that the interacting devices do not belong to the same person. Rather than introducing yet another operation, the existing *setexport* operation was extended to take an additional (optional) parameter, indicating the persons for which the file should be made visible. In our prototype implementation people are identified via unique UUID values, and devices provide the UUID of their owner via the Home protocol[11]. A special value is reserved to indicate “any person”; files exported using this value become accessible to all devices. Fully automated yet controlled file dissemination can thus be easily achieved between devices that belong to different people: the provider needs to export the desired files, and the receiver must run in the background a simple program in the spirit of Listing 3.

---

[11] This work does not address the various security issues that arise in this context.

```
void notifier( char *fname, char *location ) {
 printf( "file %s was found\n", fname );
 printf( "let's copy it locally!\n" );
 fetch( location, fname, "/mypicts/" , 60*5);
}

int ld;

ld = registerLookup( "*.jpg", notifier );
…
unregisterLookup( ld );
```

Listing 3. Registering a lookup task for fetching remote .jpg files found

7 Putting the Pieces Together

Returning to the scenario given in the beginning of this article, we outline how it can be realized using the features of the 2WEAR system. Figure 9 depicts the corresponding interactions that (may) take place between the various devices, applications, subsystems and services. The communication and discovery subsystem is not shown to avoid cluttering the picture.

Peripheral devices, like the GPS, the wristwatch and the table-display (emulated on a laptop), make their particular resources available to other devices in the form of corresponding services, such as GPSLocation, TextDisplay, Button and GraphicsDisplay.

Similarly, the camera (emulated on a PDA) exports a GraphicsDisplay and Menu service. In addition, it features a local photo application that exploits the storage management subsystem to back up new images on the repository, offload old images on

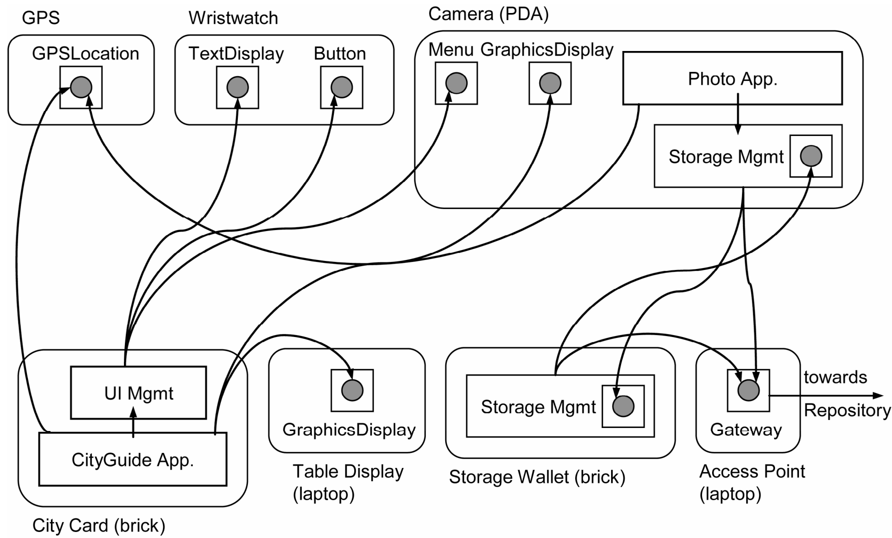

Fig. 9. Distribution of functionality and main interactions for our scenario

the user's storage wallet (brick), or let images be transferred to a storage wallet (brick) of another person. The application also periodically searches for a GPSLocation service which, if found, is used to annotate the photos taken with geographical coordinates.

The city-guide card (brick) comes with an embedded application that takes advantage of the user interface management subsystem. As a consequence, the application can be controlled via the wristwatch and/or camera user interface elements. It is possible to switch between the two configurations at runtime, via user selection or automatic adaptation (when a device is switched off or goes out of range). The map of the city can be optionally rendered on any available GraphicsDisplay service, under user control. Finally, the city-guide application employs a GPSLocation service to track user movement and issue a notification when approaching a landmark.Of course, this scenario and proposed functional arrangement is merely indicative; and by no means optimal in terms of device affordances or user interaction. We nevertheless believe that it illustrates the potential of the envisioned system as well as the level of application support that has been implemented to simplify development in such a setting.

8 Summary

In the 2WEAR project we have explored the paradigm of cooperative multi-device personal computing, where different wearable, portable and infrastructure elements communicate with each other in an ad-hoc fashion. The physical decoupling of the elements that form the system results in great flexibility, making it possible for the user to change its configuration in a simple way and at any point in time. However, it is precisely due to this dynamic nature of the system that makes the exploitation of the distributed available resources difficult. An additional challenge is to achieve this without forcing the user to (continuously) provide input to the various devices of the system in an explicit manner. Towards this objective, advanced mechanisms were developed on top of an open communication and discovery framework, which address key issues of distributed user interface and storage management on behalf of the application programmer.

For the interested reader, a more elaborate introduction of the syntax-based protocols is given in (Gutknecht 2003). The UI framework is described in detail in (Savidis and Stephanidis 2005a; Savidis and Stephanidis 2005b). Finally, work on different aspects of core runtime support for personal area network computing and a more updated version of the storage management facility is reported in (Karypidis and Lalis 2005) and (Karypidis and Lalis 2006), respectively.

Acknowledgements

This work was funded in part through the IST/FET program of the EU, contract nr. IST-2000-25286.

References

Amft, O., Lauffer, M., Ossevoort, S., Macaluso, F., Lukowicz, P., Troester, G.: Design of the QBIC wearable computing platform. In: Proceedings 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Texas, USA, IEEE, Los Alamitos (2004)

Abowd, G., Mynatt, E.: Charting Past, Present, and Future Research in Ubiquitous Computing. ACM Transactions on Computer-Human. Interaction 7(1), 29–58 (2000)

Bang, W., Chang, W., Kang, K., Choi, W., Potanin, A., Kim, D.: Self-contained Spatial Input Device for Wearable Computers. In: Proceedings7th IEEE International Symposium on Wearable Computers, NY, USA, pp. 26–34. IEEE Computer Society Press, Los Alamitos (2003)

Bhagwat, P.: Bluetooth: Technology for Short-Range Wireless Applications. IEEE Internet Computing 5(3), 96–103 (2001)

Beck, J., Gefflaut, A., Islam, N.: MOCA: A Service Framework for Mobile Computing Devices. In: Proceedings 1st ACM International Workshop on Data Engineering for Wireless and Mobile Access, Seattle, USA, pp. 62–68. ACM Press, New York (1999)

Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - A Component System for Pervasive Computing. In: Proceedings 2nd IEEE International Conference on Pervasive Computing and Communications, Florida, USA, pp. 67–76. IEEE Computer Society Press, Los Alamitos (2004)

Brumitt, B., Meyers, B., Krumm, J., Kern, A., Shafer, S.: EasyLiving: Technologies for Intelligent Environments. In: Thomas, P., Gellersen, H.-W. (eds.) HUC 2000. LNCS, vol. 1927, pp. 12–29. Springer, Heidelberg (2000)

Calvary, G., Coutaz, J., Thevenin, D., Rey, G.: Context and Continuity for Plastic User Interfaces. In: Proceedings I3 Spring Days Workshop on Continuity in Future Computing Systems, Porto, Portugal, CLRC, pp. 51–69 (2001)

Fishkin, K.P., Partridge, K., Chatterjee, S.: Wireless User Interface Components for Personal Area Networks. IEEE Pervasive Computing 1(4), 49–55 (2002)

Gajos, K., Christianson, D., Hoffmann, R., Shaked, T., Henning, K., Long, J.J., Weld, D.: Fast And Robust Interface Generation for Ubiquitous Applications. In: Proceedings 7th International Conference on Ubiquitous Computing, Tokyo, Japan, pp. 37–55 (2005)

Garlan, G., Siewiorek, D., Smailagic, A., Steenkiste, P.: Project Aura: Toward Distraction-Free Pervasive Computing. IEEE Pervasive Computing 1(2), 22–31 (2002)

Gellersen, H.G., Schmidt, A., Beigl, M.: Multi-Sensor Context Awareness in Mobile Devices and Smart Artifacts. Mobile Networks and Applications 7(5), 341–351 (2002)

Gutknecht, J.: A New Approach to Interoperability of Distributed Devices. In: Stephanidis, C. (ed.) Universal access in HCI: inclusive design in the information society, pp. 384–388. Lawrence Erlbaum, Mahwah (2003)

Johanson, B., Fox, A., Winograd, T.: The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing 1(2), 67–74 (2002)

Karypidis, A., Lalis, S.: Exploiting co-location history for efficient service selection in ubiquitous computing systems. In: Proceedings 2nd International Conference on Mobile and Ubiquitous Systems, San Diego, USA, pp. 202–209. IEEE Computer Society Press, Los Alamitos (2005)

Karypidis, A., Lalis, S.: Omnistore: A system for ubiquitous personal storage management. In: Proceedings 4th IEEE International Conference on Pervasive Computing and Communications, Pisa, Italy, pp. 136–146. IEEE Computer Society Press, Los Alamitos (2006)

Kortuem, G., Schneider, J., Preuitt, D., Thompson, T.G.C., Fickas, S., Segall, Z.: When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad-Hoc Networks. In: Proceedings 1st International Conference on Peer-to-Peer Computing, Linkoping, Sweden, pp. 75–94 (2001)

Krause, A., Siewiorek, D., Smailagic, A., Farrington, J.: Unsupervised, Dynamic Identification of Physiological and Activity Context in Wearable Computing. In: Proceedings 7th IEEE International Symposium on Wearable Computers, NY, USA, pp. 88–97. IEEE Computer Society Press, Los Alamitos (2003)

Kuladinithi, K., Timm-Giel, A., Goerg, C.: Mobile Ad-hoc Communications in AEC Industry. ITcon 9(Special Issue on Mobile Computing in Construction), 313–323 (2004)

Kunito, G., Sakamoto, K., Yamada, N., Takakashi, T.: Architecture for Providing Services in the Ubiquitous Computing Environment. In: Proceedings 6th International Workshop on Smart Appliances and Wearable Computing, Lisbon, Portugal (2006)

Lehikoinen, J., Holopainen, J., Salmimaa, M., Aldrovandi, A.: MEX: A Distributed Software Architecture for Wearable Computers. In: Proceedings 3rd International Symposium on Wearable Computers, Victoria, Canada, pp. 52–57. IEEE Computer Society Press, Los Alamitos (1999)

Majoe, D.: Ubiquitous-computing enabled wireless devices. In: Stephanidis, C. (ed.) Universal access in HCI: inclusive design in the information society, pp. 444–448. Lawrence Erlbaum, Mahwah (2003)

Ogris, G., Stiefmeier, T., Junker, H., Lukowicz, P., Troester, G.: Using Ultrasonic Hand Tracking to Augment Motion Analysis Based Recognition of Manipulative Gestures. In: Proceedings 9th IEEE International Symposium on Wearable Computers, Osaka, Japan, pp. 152–159. IEEE Computer Society Press, Los Alamitos (2005)

Park, D.G., Kim, J.K., Bong, S.J., Hwang, J.H., Hyung, C.H., Kang, S.W.: Context Aware Service Using Intra-body Communication. In: Proceedings 4th IEEE International Conference on Pervasive Computing and Communications, Pisa, Italy, pp. 84–91. IEEE Computer Society Press, Los Alamitos (2006)

Pascoe, R.: Building Networks on the Fly. IEEE Spectrum 38(3), 61–65 (2001)

Roman, M., Cambell, R.H.: Gaia: Enabling Active Spaces. In: Proceedings 9th ACM SIGOPS European Workshop, Kolding, Denmark (2000)

Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: Proceedings ACM SIGCHI 99 Conference on Human Factors in Computing Systems, Pittsburgh, USA, pp. 434–441. ACM Press, New York (1999)

Satyanarayanan, M.: The Evolution of Coda. ACM Transactions on Computer Systems 20(2), 85–124 (2002)

Savidis, A.: Supporting Virtual Interaction Objects with Polymorphic Platform Bindings in a User Interface Programming Language. In: Guelfi, N. (ed.) RISE 2004. LNCS, vol. 3475, pp. 11–22. Springer, Heidelberg (2005)

Savidis, A., Stephanidis, C.: Dynamic deployment of remote graphical toolkits over Bluetooth from wearable devices. In: Proceedings 11th International Conference on Human-Computer Interaction, Las Vegas, USA (2005a)

Savidis, A., Stephanidis, C.: Distributed interface bits: dynamic dialogue composition from ambient computing resources. Personal Ubiquitous Computing 9, 142–168 (2005b)

Schroeder, M.D., Gifford, D.K., Needham, R.M.: A caching file system for a programmer's workstation. SIGOPS Operating Systems Review 19(5), 25–34 (1985)

Tuulari, E., Ylisaukko-Oja, A.: SoapBox: A Platform for Ubiquitous Computing Research and Applications. In: Proceedings International Conference on Pervasive Computing, Zurich, Switzerland, pp. 125–138 (2002)

DeVaul, R., Sung, M., Gips, J., Pentland, A.: MIThril 2003: Applications and Architecture. In: Proceedings 7th IEEE International Symposium on Wearable Computers, NY, USA, IEEE Computer Society Press, pp. 4–11. IEEE Computer Society Press, Los Alamitos (2003)

Want, R., Pering, T., Danneels, G., Kumar, M., Sundar, M., Light, J.: The Personal Server: Changing the Way We Think About Ubiquitous Computing. In: Borriello, G., Holmquist, L.E. (eds.) UbiComp 2002. LNCS, vol. 2498, pp. 194–209. Springer, Heidelberg (2002)

Weiser, M.: The Computer for the 21st Century. Scientific American 265(3), 94–104, reprinted in IEEE Pervasive Computing 1(1), 19–25 (1991)

Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Communications of the ACM 20(11), 822–823 (1977)

Yasuda, K., Hagino, T.: Ad-Hoc Filesystem: A Novel Network Filesystem for Ad-hoc Wireless Networks. In: Lorenz, P. (ed.) ICN 2001. LNCS, vol. 2094, pp. 177–185. Springer, Heidelberg (2001)