Distribution:	public ☑	restricted ☐	confidential ☐

INFORMATION SOCIETIES TECHNOLOGY (IST) PROGRAMME

PROJECT 2WEAR

A Runtime for Adaptive and Extensible Wireless Wearables

IST-2000-25286

Report on
Design of Core Runtime

Deliverable No.:	*D6*
Due date:	*31.3.2002*
Delivery date:	-
Description:	*This document describes the design of the core runtime for a 2WEAR system.*
Partners owning:	*ETHZ*
Partners contributed:	*ETHZ, FORTH, NOKIA*
Made available to:	*Public.*

Table of Contents

1 COMPONENT ARCHITECTURE

1.1 The general component model

A *component* is a logical participant in the distributed system consisting of wearable devices, portable devices and the (Internet) backbone. In particular, every individual wearable device of any complexity is a component, as is any self-contained functional unit of a device. Components can be statically or dynamically composed into *composites* that, in principle, are simply components on a next hierarchical level. For example, any ad hoc group of wearable devices can act as a single 2WEAR component.

In general, a component takes both an active role and a passive role, it is an autonomous *actor* developing in time according to some intrinsic law of behavior, as well as a *shared resource* offering specific services to its environment.

1.2 Interfaces

In 2WEAR, *interfaces* represent services or, more exactly, abstract "contract" views of services. In the following, we propose a generic, syntax-based formalism that is suitable for the representation of a wide spectrum of distributed interfaces. However, considering interoperability a crucial issue in 2WEAR, additional research is definitely needed before any interface model can be finalized.

According to our proposal, each interface is defined by a *formal syntax* to be interpreted by the implementation of the corresponding service. In the following examples, EBNF (Extended Backus Naur Form) is used as a notation, where symbols in normal and **bold** face respectively mean message traffic from caller to callee and **from callee to caller** respectively.

Examples of different kinds of interfaces are given below:

Command list

```
Player = "start" | "play" | "stop" |  "reset".
```

Property manager

```
PropMan =  "get" prop value | "set" prop value | "enum" { prop } "~".
prop = String.
value = float | string.
```

Clock with keyword parameters

```
Clock = "getTime" value value | "setTime" [ "d" value ] "t" value.
value = integer.
```

Event source
```
EventInterface = "register" uid interface.
uid = integer.
interface = string.
```

Query for proposal
```
Proposal = "get" { proposal "nok" } proposal "ok".
proposal = string.
```

Streaming (with known length)
```
StreamControl = "get" stream max len { byte }.
stream = string.
max, len = integer.
```

Streaming (with unknown length)
```
StreamControl = "get" stream terminator { byte } terminator.
stream = string.
terminator = byte. (that does not appear in the byte stream)
```

The following table of correspondence fixes some terminology. Obviously, the mapping of terminal syntactical symbols such as *integer*, *float*, *string* etc. into sequences of bits must be defined explicitly and universally.

SYNTAX VIEW	INTEROPERABILITY VIEW
Terminal symbol	Data type
Any symbol	Message
Sentence	Call

We call an interface *I1* a *refinement* of interface *I* if and only if the syntax of *I1* extends the syntax of *I*. A typical step of refinement is adding commands ("methods") to a "method table" interface. However, our refinement concept is much more general. For example, it includes extensions such as adding parameters to a command and adding type variants to a parameter. If a component implements a refinement *I1* of an interface *I* then, by convention, it implicitly implements *I* as well.

Interfaces are identified uniquely by their name within a global hierarchy of *namespaces*. In any absolute context, interface names should be qualified explicitly by their namespace.

1.3 Components

Components present themselves uniformly to their environment in the form of a.) a *universal identifier (uid)* and b.) a set of interfaces.

Each uid uniquely identifies a specific instance of a component. While the uid of components corresponding to devices are assigned statically, uids of ad hoc components such as, for example, individual functional parts or logical groups of devices, must be generated on the fly.

The set of interfaces presented by a component subsumes its functionality. A full implementation must be provided by the component for every interface it exposes.

1.4 The Home interface

Each component participating in a 2WEAR environment must at least implement the *home interface* that provides some basic functionality, at least comprising:

- Ability of other (remote) components to announce their existence (to this component)
- Enumeration of interfaces implemented by this component
- Self-documentation of this component

In EBNF, the home interface could look like this:

```
HomeInterface = "announce" uid  |  "getallifc" { interface } "~" |
                "getifc" selexpr { interface } "~" | "getdoc" docTxt.
uid = integer.
selexpr, interface, docText = string.
```

In passing we note that interface descriptions in EBNF can easily be transformed into other formalisms. For example, an XML version of the home interface could look like this:

```
<INTERFACE TYPE="HOME" NAME="X">
  <METHOD NAME="register"><PAR TYPE=in>integer</PAR></METHOD>
  <METHOD NAME="getallifc"><PAR TYPE=out>string[]</PAR></METHOD>
  <METHOD NAME="getifc"><PAR TYPE=in>string</PAR>
                        <PAR TYPE=out>string[]</PAR></METHOD>
  <METHOD NAME="getdoc"><PAR TYPE=out>string</PAR></METHOD>
</INTERFACE>
```

The home interface has two prominent uses. The first is notifying the component of the existence of new participating components in the 2WEAR arena. The second is querying the component for the services it provides.

1.5 Communication between components

Components communicate with each other via syntax-controlled protocols based on a *master/slave* message exchange paradigm. Conceptually, the caller (master) and the callee (slave) run separate sequential processes that synchronize and communicate via *Send/Receive* primitives. Messages (including binary data) are syntactical entities that can be sent by either partner and received by its companion. In case buffering is available at the

receiving end, a Send operation need not block the sender. In contrast, Receive operations always block the issuing partner until the data has been received completely.

Note that the companion process is implicit in the case of the callee, while it must be specified on the caller side. Also note that ordinary (blocking) method calls can be modelled in our framework as follows:

Procedural without parameter
```
Caller = { Call f }
Callee = { Run implementation of f }
```

Procedural with input parameter
```
Caller = { Call f; Send(x) }
Callee = { Receive(x); Run implementation of f }
```

Functional with input parameter and result
```
Caller = { Call f; Send(x); Receive(y) }
Callee = { Receive(x); Run implementation of f; Send(y) }
```

1.6 Components and composition

Components can be implemented by hardware devices either on a n to 1 basis (for example, components represented by some functional unit of a server system), on a 1 to 1 basis (for example, components represented by wearable devices) or on a 1 to n basis (for example, components represented by a logically connected group of wearable devices). In the latter case, the group is typically represented by one of its members that takes the role of a *coordinator*. In such a case, the coordinating device represents both itself and the group it coordinates. In particular, it coordinates and synchronizes the group's intrinsic behavior with the handling of interface calls.

1.7 Implementation of components

Basically, components can be implemented in any arbitrary development environment and in any programming language. However, each implementation must a.) comply with the interoperability framework outlined above and b.) provide an implementation of the 2WEAR core runtime API. This API in particular includes facilities such as *profiling* and *service registry*, as well as functional blocks for *external communication*, *service discovery* and *service management*. The 2WEAR core runtime API will be described briefly in section 3.

2 COMMUNICATION TECHNOLOGY

2.1 Wireless Protocols

Communication protocols are essential ingredients of any distributed system. In the case of wireless connectivity, the following issues are of particular importance:

- Efficiency
- Low power use
- Broadcast nature of signal

Efficiency is achieved by using an appropriate encoding scheme that keeps overhead data at a minimum. Compression can additionally be used but the typically low performance of the micro controllers involved must be taken into consideration. The broadcast nature of the wireless signal can be used at a low level for simple and fast device discovery.

2.2 The 2WEAR Communication Architecture

As depicted in Figure 1, a generic 2WEAR system roughly comprises three kinds of participants: *Portable* or *wearable devices*, *gateways*, *Internet backbone*.

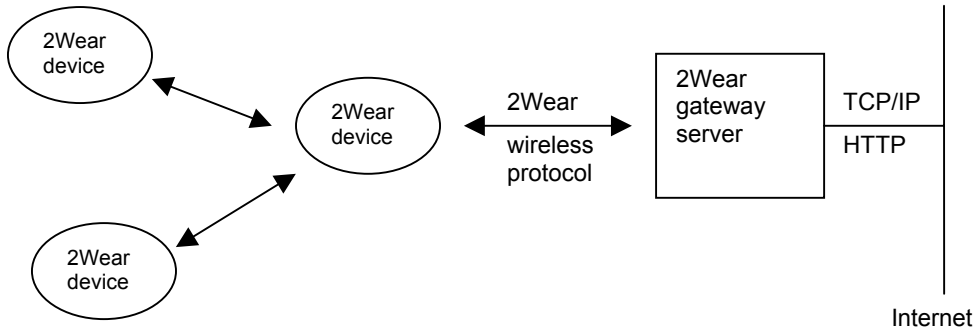

Fig.1 The Internet gateway makes Internet contents available to 2WEAR devices.

The 2WEAR gateways' primary task is translating the TCP/IP protocol stack (including Internet RFC) into the 2WEAR protocol and vice versa. Thanks to the availability of various communication device drivers and TCP/IP implementations, Windows CE and/or Linux are promising platforms for 2WEAR gateways to be implemented on.

2.3 Bluetooth Radio

The Bluetooth specification includes both link layer and application layer definitions for data, voice and content-centric applications. Radios complying with this specification operate within

the unlicensed 2.4 GHz radio spectrum, which ensures communication compatibility worldwide. These radios use a spread spectrum, frequency hopping, full-duplex signal at up to 1600 hops/sec. The signal hops among 79 frequencies at 1 MHz intervals to give a high degree of interference immunity.

2.4 The 2WEAR ”Bluetooth-Lite” Protocol

While it was agreed among the 2WEAR partners on using *Bluetooth* as a communication medium, the full Bluetooth protocol stack was considered less-than-suitable and too heavyweight for the envisaged 2WEAR scenarios.

Current plans therefore aim at implementing a lightweight protocol and interfacing it with the Bluetooth radio layer via HCI. This will allow the as-is reuse of two powerful built-in Bluetooth services:

- Fast acknowledgement and frequency hopping
- Device detection via broadcast

Datagrams of the 2WEAR communication protocol will consist of a *header* followed by a sequence of *data bytes*. The header could include the following control information:

- Sender uid
- Receiver uid
- Data type
- Data length

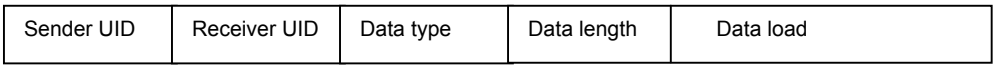

Fig.2 The 2WEAR wireless protocol header plus data load.

Both the sender uid and the receiver uid uniquely identify a 2WEAR physical radio. The data type is used to identify messages belonging to different (higher level) protocols, while the data length is used to indicate the length of the following data load. The latter is protocol dependent, e.g. typically contains the uid of the recipient component followed by a list of parameters.

3 CORE RUNTIME ARCHITECTURE

3.1 The Core Runtime API

As mentioned in section 1.7, the core runtime system presents itself to the applications as a comprehensive API structured into functional blocks. The following blocks have been identified so far:

Profiling

Provides access to component profile and/or user profile defined in XML-like notation.

External communication

Supports calling of and listening to external components. The simplest scenario to be supported is *mimicked RMI*. The corresponding API would take input parameters and return output parameters and it would block the caller while sending, receiving and processing takes place. If process scheduling is integrated with the external communication service, the scheduler can obviously pass control to any other process in the ready pool while blocking a caller. It is worth noting, however, that our interface communication model allows the communication service to support much more complex, asynchronous scenarios. For example, a calling process could register for the reply message and proceed with another task until the reply arrives.

Service registry

Register containing the entirety of interfaces implemented by this component.

Service discovery

Passively and actively discovers interfaces provided by this component and by its current environment.

Service management

Provides an abstract class for service objects to be stably linked to application objects. Provides basic fault tolerant functionality for acquiring, connecting and switching concrete devices and media that are suitable for delivering the desired service. *User interface elements* are an important subset of service objects.

3.2 An Implementation Based On Active Objects

The goal of the *Active Objects* project at ETHZ is the design and efficient implementation of a universal and unified runtime model for concurrent “active” objects with some intrinsic

behavior, modelled as a separate process. The project involves a programming language, a compiler and a run-time system.

The *Active Oberon Language* is an evolution of Pascal, Modula and Oberon. Four new constructs support the implementation of concurrent, active objects:

- Object body expressing the object's intrinsic behavior
- Object-specific access protection
- System-managed assertions
- Pre-emptive priority scheduling

The current *Active Oberon Compiler* is a cross-compiler running on the *Native Oberon* runtime. It itself uses the metaphor of active objects by launching a separate active parser object for each scope to be compiled within an Active Oberon module.

The *Active Object Support* (AOS) is a compact kernel that serves as a supportive runtime host for active objects. It currently runs directly on any *Intel SMP* hardware platform. In the context of the 2WEAR project, a *StrongARM* version targeted at PDAs and wearable devices such as the *PCB StrongARM* board is under development.

The *Bluebottle* operating system consists of the Active Object Support kernel plus a dynamic collection of modules that provide generic abstractions for devices and services such as, for example, file systems, user interfaces and TCP/IP networking.

3.3 An Implementation on Linux

In addition to the AOS implementation, some 2WEAR modules/functional blocks will be implemented on top of the Linux platform. The main reasons for this 'dual' approach are:

- De-couple the work performed at the level of the runtime system (AOS) from the work performed at the level of 2WEAR functional blocks (on top of Linux)
- Allow partners (besides ETHZ) to develop and test functional blocks of the 2WEAR system using familiar programming languages and environments, e.g. C, C++ and Java
- Verify, within the project, that the 2WEAR protocols are platform independent
- Provide implementations that run on a system (Linux) that is widely used within the academic and industrial community, but still experiment with a small and efficient system (AOS) that is more appropriate for wearable and resource-constrained devices

Once implementations for the various 2WEAR modules are available on either platform, it is expected that they can easily be ported on top of the other. Moreover, several public domain implementations that exist for Linux can be exploited to accelerate development in AOS.