VTT INFORMATION TECHNOLOGY

SPECIFICATION
TEKES contract no: 40501/01

Wireless Wellness Monitor II (WWM II)

Software Architecture

Version 1.3

21.5.2002

Timo Tuomisto, Lasse Pekkarinen, Luc Cluitmans,
Ilkka Korhonen

VTT

Version history

Version	Date	Author(s)	Reviewer	Description
0.1	2001-10-10	TTu		draft of contents
0.2	2001-10-16	TTu, LPe	VTT WWM2 workgroup	more stuff in Location, Proximity, OSGi spec 2.0 revisions
0.3	2001-10-22	LPe, TTu, LCl		suggestions after check
0.4	2001-10-22	TTu, LPe		inclusion of some software components
0.5	2001-11-19	TTu		fixed several typing errors, duplicate text
0.9	2002-01-09	IKo		Clarification of presentation
1.0b	2002-01-18	TTu, LPe		Revisions on databases, Proximity, TV
1.1	2002-04-03	LPe		added appendix 3, checked from start until a new section 5.1.3 JES dependency & persistence, architecture figures
1.2c	2002-04-05	LPe		checked and fixed from 5.1.3, according to the implementation
1.2d	2002-04-08	TTu		Clarification of text, rearranging problems encountered to Appendix, added some References
1.3	2002-05-21	IKo		Final version

Contact information

Ilkka Korhonen
VTT Information Technology
P.O. Box 6, FIN-33101 Tampere, Finland
Street Address: Sinitaival 6, Tampere
Tel. +358 3 316 3111, fax +358 3 317 4102
Email: ilkka.korhonen@vtt.fi
Web: http://www.vtt.fi/tte/

Contents

1. Introduction

This document provides the main architectural principles and guidelines for developing the WWM II software. It also provides a list problems encountered, and some deviations in the final implementation from the original architectural design. The details of object classes and interfaces are described in documentation generated by Javadoc.

The hardware for the WWM II demonstration system is introduced in [1] and the functional requirements are introduced in [2].

2. Design principles

The WWM II demonstration system consists an OSGi-based home server, which communicates with IP-based devices or device aggregates (networks). The home server can be accessed by an HTML browser.

The home server is an implementation of OSGi release 1.0 [3], and is based on JES 2.0 (an implementation of OSGi release 1) with some custom extensions. Whenever appropriate, the custom extensions are aligned with the OSGi release 2 specification, which has recently appeared. The home server is based on Java JVM 1.3 or later.

The non-IP devices (mostly RS232 protocol based) and device aggregates have a front-end proxy, which makes them IP based.

All concepts - devices, persons or locations – are introduced as *services* registered to the OSGi framework. Hence, the services registered to the framework include each individual user (`Person` service), each location (`Location` service), each user display (`Display device` service), and each individual peripheral device. This is done to have a uniform search mechanism for all services found in the framework, and make references between them.

3. Architecture description

3.1. Introduction

The Open Service Gateway initiative (OSGi) is a group of organizations to develop a platform for integrating the control and use of home appliances. The Open Service Gateway Specification release 1.0 [3] is the architectural backbone for building the WWM II demonstration system. An overview of the framework is given in [4].

OSGi defines a set of interfaces (framework) to manage the access and lifecycle of the services. The specification also includes a minimum standard set of service interfaces (http, log, device etc.) for OSGi compliant implementations to conform. Moreover, there are some implicit functional requirements for the OSGi compliant systems to obey, e.g. the inclusion of a Device Manager, which is responsible of acquiring and installing drivers for any newly detected devices.

Java Embedded Server (JES) [5] (enhanced with patch [6]) provides an implementation for OSGi. It also enhances the OSGi with some new functionality e.g. with a set of interfaces and implementation to manage users and their authentication in the system. Already in the JES implementation a few extra rules have been added to allow at least minimum integration of services within Home Portal, e.g. the parameters used to register a servlet to the `ServletContext` is a JES internal feature.

Recently, the OSGi specification release 2 [7] has appeared. It refines, enhances and modifies many features found in release 1. Some cornerstone modifications of release 2 are embedded in this document. The direct loans from that document are written with Font Size 10.

The WWM II architectural basis is still OSGi release 1, but some changes suggested by OSGi release 2 have been adopted. The WWM II User Scenarios [2] and Physical Network Specification [1] introduce some new concepts, which neither the OSGi specification v1.0 nor the corresponding JES implementation issue at all. In the following the concept enhancements to release 1 are introduced with possible guidelines to implement them by OSGi release 2.

3.2. Conceptual schema (ontology) and registered services

When artificial intelligence is built into a software environment ontology plays a major role as the intelligent agents function based on their world view. The conceptual model of OSGi standard has been kept in minimum.

The ontology requirements set by the WWM II user scenarios [2] are minor, because most of the functionality within agents is done by intelligent functions, which are acting upon a single service. Hence, the current WWM II system does not require conceptual trees for artificial reasoning system. There is also no need to categorize the messages shown to the users so that an agent capable of reasoning would filter and show only important messages[1]. Appendix 2 provides an example of a complex ontology model. It is very unlikely that such a world view can be standardized in a large scale.

However, there were some conceptual additions that needed to be introduced for the WWM II. A sufficient conceptual schema for WWM II works with the interfaces provided by services registered into OSGi. Most important roles in combinatory usage have concepts such as person, proximity, display device, application, home portal, health database, and electricity socket.

[1] Though in the user trials a need for this kind of filtering appeared clearly.

3.3. Proximity

The WWM II project adds the notion of proximity to the OSGi framework. In the Physical Network Specification [1] there are several physical means to detect the proximity of persons, displays, locations, and devices to each other. The WWM II system utilizes this information to improve the management of home environment, and especially the management of the environment in the proximity of a person, when he is moving around at home.

3.4. Location

The proximity concept brings another issue of space, namely locations (house, living room, kitchen etc.). The information that the user or a display device are close to a certain location in the house may be utilized in assisting the user to select, or remind about topics, which are **spatially relevant.** If the system knows that we are in the proximity of kitchen it may reason that we also must be at home. This requires some structural knowledge of locations.

NOTE: The WWM II system doesn't use absolute locations, i.e. the exact space coordinates of the services. However, if introduced, these services can be used via Location and Proximity interfaces, which are reasonable abstractions or derivatives of position information.

3.5. User Interfaces

Traditionally the interaction between a user and a system happens via an information appliance, like PC, PDA, mobile phone etc. However, the use of physical devices having an inherent physical "user interface" as well extends the user interface from being a traditional display device to include the "user interactions with the other physical (not only information appliances) devices as well. The software architecture has to be partly turned around: in addition to (the users of) display devices being capable of controlling the devices, also the devices know the existence of display devices, which they can also control when the user interacts with the device user interfaces.

3.5.1. Push

To be able to control the display devices (which are essentially browsers) in the HTML view model, a push mechanism must be incorporated to allow the display device to be manipulated by the system. However, while implementing and using this option care must be taken that the user feels he still has the control on the system. If the display device is an HTML browser, proprietary applets may be used to build push.

As explained in [2] it is assumed that PDAs are private user devices, and the information displayed on them may be kept in discretion. In turn, the TV is a public device, which is probably visible for a large audience. Therefore the system have to

allow a mechanism to block any discrete information from popping up in a **public display device**.

3.6. Person

The concept of person is not handled at all in release 1 of OSGi specification. Each user of WWM II will be associated with a persistent `Person` service. Following the ideas of OSGi release 2, we give an unique `service.pid` to each person, so that it is possible to have person related persistent data in section 3.7.5.

UserAdmin [release 2 of OSGi]

The package `org.osgi.service.useradmin` includes user authentication and authorization. It also introduces a `user` object to store persistent data.

3.7. Persistency

Persistency is partly orthogonal for the conceptual model objects. It is something all the services or other objects either have or not, and the persistency mechanism should be invisible at the registered services level of OSGi.

3.7.1. The persistency problem in OSGi release 1

The OSGi specification 1.0 and its implementations have a inherent serious deficiency; The `BundleContext.registerService`(), which is used to register services to the framework for public use between bundles, has no internal persistency mechanism. When shutting down the framework, each bundle may store in an appropriate manner internally the services it has registered to framework. Otherwise the registered services disappear from the framework. On restart of the framework the stored services will have to be re-registered. However, re-registering creates always a new `ServiceRegistration` with new `ServiceReference`s, which are unequal to the previously registered services as understood by

`ServiceReference.equals`(ServiceReference previous)

Thus ServiceReference cannot as such be used as a persistent reference. Inside bundles the persistent reference problem between services can be handled in an inherently appropriate manner. However, if persistent references to services in other bundles are required, the release 1.0 misses its target: the `ServiceReference` objects don't carry any persistent information, which could be used to identify the original service when (after shutdown) reregistering to the framework.

WWM II has several persistent configurable references to other services between bundles and thus a Persistency mechanism of persistent bundle service references is required.

Persistent identity [OSGi release 2]

In chapter 9.3 of [7] the persistent identity is discussed. WWM II adapts the notation of using service.pid as persistent identifier: all services requiring persistency should be registered with a property `service.pid`, the name of which is defined in org.osgi.framework.Constants.SERVICE_PID. When a bundle registers (or re-registers) as a persistent service it should always use the same PID. The PID should be unique at least in their OSGi framework implementation.

3.7.2. The configuration admin service specification and OSGi release 2

Whatever the physical device behind the OSGi gateway, it needs some kind of persistency mechanism to store device specific configuration data. In OSGi release 2 the framework is enhanced with a set of packages to tackle the persistency issues of the configuration of the services. The `ConfigurationAdmin` service maintains a persistent database of `Configuration` objects locally or remote, that are registered with a `service.pid` –property and implement one of the two interfaces:

- *Managed Service* -A service registered with this interface receives its configuration dictionary from the database or null
- *Managed Service Factory* –Service Registered with this interface receive several configuration dictionaries when registered

The `ConfigurationAdmin` service is intended to replace the `Configurable` interface found in release 1. One can use the metatype package of OSGi release 2. The reasons for dropping the `Configurable` interface are discussed in chapter 9.12 of release 2 [7].

3.7.3. The preferences service specification and OSGi release 2

The package `org.osgi.service.prefs` is another persistency extension to store persistent personal data or some other service related data (game highest score etc.). This package allows bundles to store and retrieve properties stored in a tree of nodes, where each node implements the Preferences interface. The preferences interface allows a bundle to create or obtain a Preferences tree for bundle properties as well as for each user of bundle.

The Preferences Service does not provide a mechanism to allow one bundle to access the preferences data of another. If a bundle wishes to allow another bundle to access its preferences data, it can pass a Preferences or Preferences Service object to that bundle. The focus of preferences specification is simplicity, not reliable access to stored data.

3.7.4. Java Data Objects

The OSGi name has been also associated with Java Data Objects (JDO) [8,9], which is a forthcoming Java proposal to build a totally object oriented persistency interface, which hides the implementation details. However, no trace of JDO is found in release 2 of OSGi spec [7].

3.7.5. WWM II Persistent Data

3.7.5..1 Configuration data

It is assumed that all configuration happens by a `ResourceBundle` file within each bundle. Before a bundle is uploaded/updated, the ResourceBundle is edited by the service provider to fit the client's needs. This makes sense because WWM II does use

device discovery only marginally (see 5.4) and the devices in the local network are found by contacting predermined addresses

Some temporary configuration may still be done to Configurable interfaces through administrative user interfaces, which manipulate the Configurable interface, but these changes are not persistent under shutdowns / restarts of the service. If persistent configuration management is required, the ResourceBundle object is edited and the containing bundle is updated to the OSGi environment.

3.7.5..2 Personal data or other measurement data

The OSGi 2 specification doesn't define any a general database service with transactions and atomicity guarantees. The `preferences` package services can be used to store persistent user related data, but it is questionable, whether this form of data storage is suitable for healthcare data e.g. weighing or activity data.

In WWM II personal data (e.g. health data) is loaded to a separate server running remotely (or locally) via SQL statements. All services can use a common database, which is configured for them in a single bundle: Persistency. Using this bundle the WWM II services get a connection to the SQL-server.

4. Software components

WWM II software components are depicted in Figure 1. In the following these components are shortly introduced.

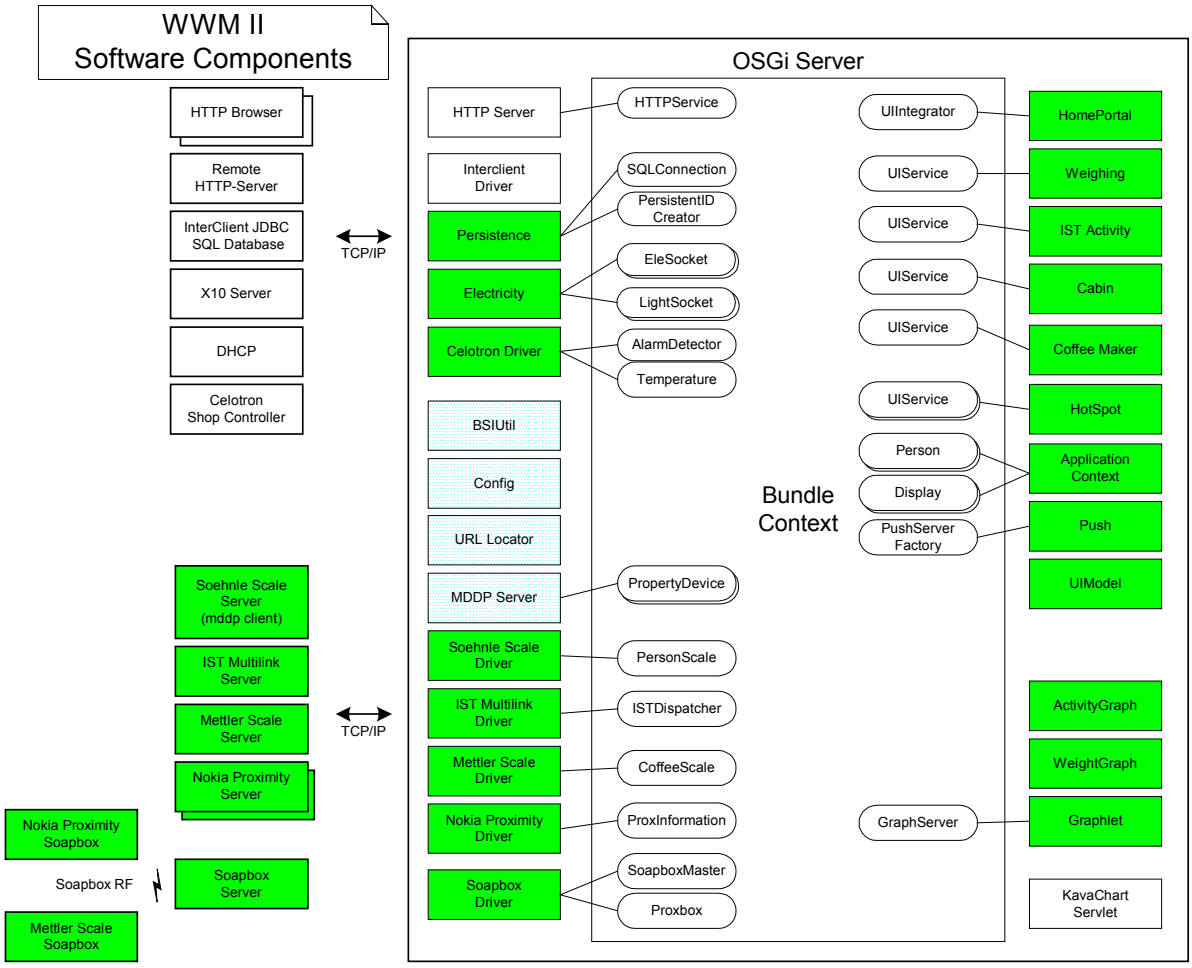

Figure 1. WWM II components. Processes are shown as rectangles with thick line. Bundles are shown as rectangles with thin line. Services registered in BundleContext are shown as ovals, and combined with lines to their origin bundles.

4.1. Processes

All processes communicate over Internet Protocol (IP). The processes included are:

- Device proxies, who change the appropriate physical devices to IP-based devices. (SoapBoxServer, SoehleScaleServer, X10Server, MettlerScaleServer, ISTMultiLinkServer, NokiaProximityServer)

- External (to WWM II) firmware (Celotron Shop Controller) or internal firmware (Nokia Proximity SoapBox, Mettler SoapBox) based IP-services
- Network basic services (DHCP, Dynamic Host Control Protocol)
- Miscellaneous other external (outside WWM II) standard services (Database Server, Remote HTTP Server, HTTP Browser)
- The OSGi server consisting of several services introduced by bundles
- The MDDPClient component in device proxies enabling the automatic driver Bundle download specified in OSGi[2]

4.2. Bundles

The bundles consist of some functional entities added and removed from the framework as an atomic whole. Bundles may introduce Java packages including classes and interfaces for other bundles to use, but especially they install objects (Services) implementing known interfaces to `BundleContext` for other Services to utilize.

The bundles included are:

- OSGi (JES) basic Bundles (HTTP Server) or some very standard extensions (WAP Gateway).
- Device Driver Bundles, which communicate with device proxies, and bring the devices behind the proxies visible (several devices if having multiplex behavior).
- Device Discovery Bundles, introducing the automatic detection of services in the Home IP Network, and obeying OSGi device discovery mechanism. (here MDDP, URLLocator, BSIUtil and Config)
- Conceptual Enhancement Bundles (Proximity, ApplicationContext, Electricity) bringing concepts outside OSGi specification to framework.
- Agent bundles (e.g. Weighing), which utilize the services provided by other bundles for additional processing.
- Utility bundles (Push, Interclient, Graphlet, GraphGenerator) for other services to use. These bundles are not architecturally mandatory, but provide some internal facilitating services.
- The Service Integration bundles (UIModel, HomePortal) used to build the home portal application from the components.

[2] Possible component, but will not be included in the demonstration system.

4.3. Services

Each bundle introduces new interface types and/or provides an implementation for interfaces and produces registered services to the framework. The services, which each bundle registers to Framework, are introduced to the system. Due to the fact that the bundles need to be installed and removed on the fly, no direct use of objects of other bundles are generally allowed but must be done via ServiceReference objects and the

```
BundleContext.getService(ServiceReference)
```

function call. Thus, all use of objects in the registered interfaces occur either by `ServiceReference`s, or by String attributes, e.g. the service.pid.

5. WWM II bundles

Figure presents another view to the WWM II software, somewhat simplified picture of the main implementation bundles at the OSGi server. In *Figure* dependencies are marked with arrows: a block uses the component below to which an arrow leads. To keep the amount of arrows small, the dependencies to UI Model are labeled with UI letters. In addition, IST Dispatcher, and all the UI-labeled bundles depend also on Application Context. The bundles are shortly described in the following sub-chapters.

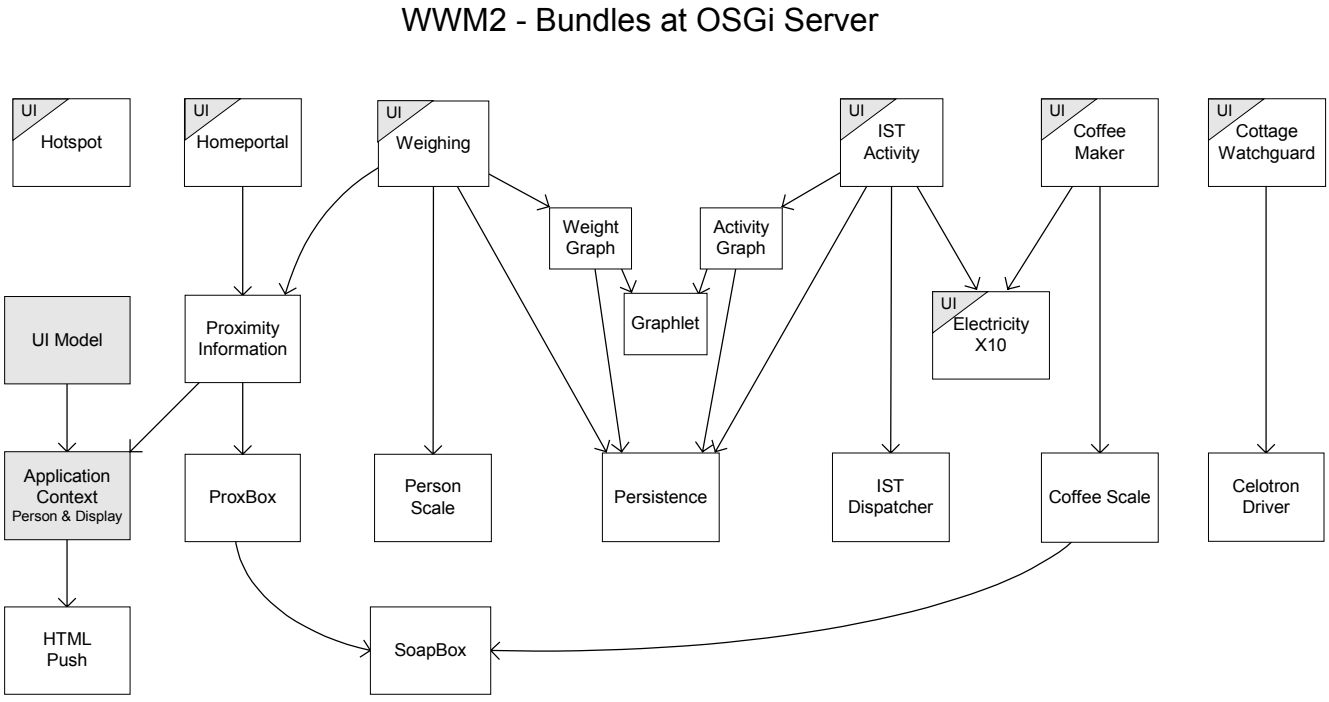

Figure 2. The WWM II bundles and their dependencies.

5.1. Spatial extensions to OSGi

The sections 5.1.1. , and 5.1.2. below provide a software design for a general proximity framework, suitable under OSGi. The section 5.1.3. describes implemented version with its simplifying ideas.

5.1.1. Proximity

The proximity bundle provides the proximity framework for other services to use. WWM II has specified the interfaces for a decent proximity bundle:

- It provides an abstracted API (Proximity) to check the spatial proximity between services, which have been associated (mapped) to any proximity sensors.
- It allows services to register themselves as `ProximityListener`s for `ProximityEvent`s .
- It provides an agreed API (`ProximityAccessPoint`) that any Sensor Types capable of detecting the proximity of each other can implement and join to the proximity framework.

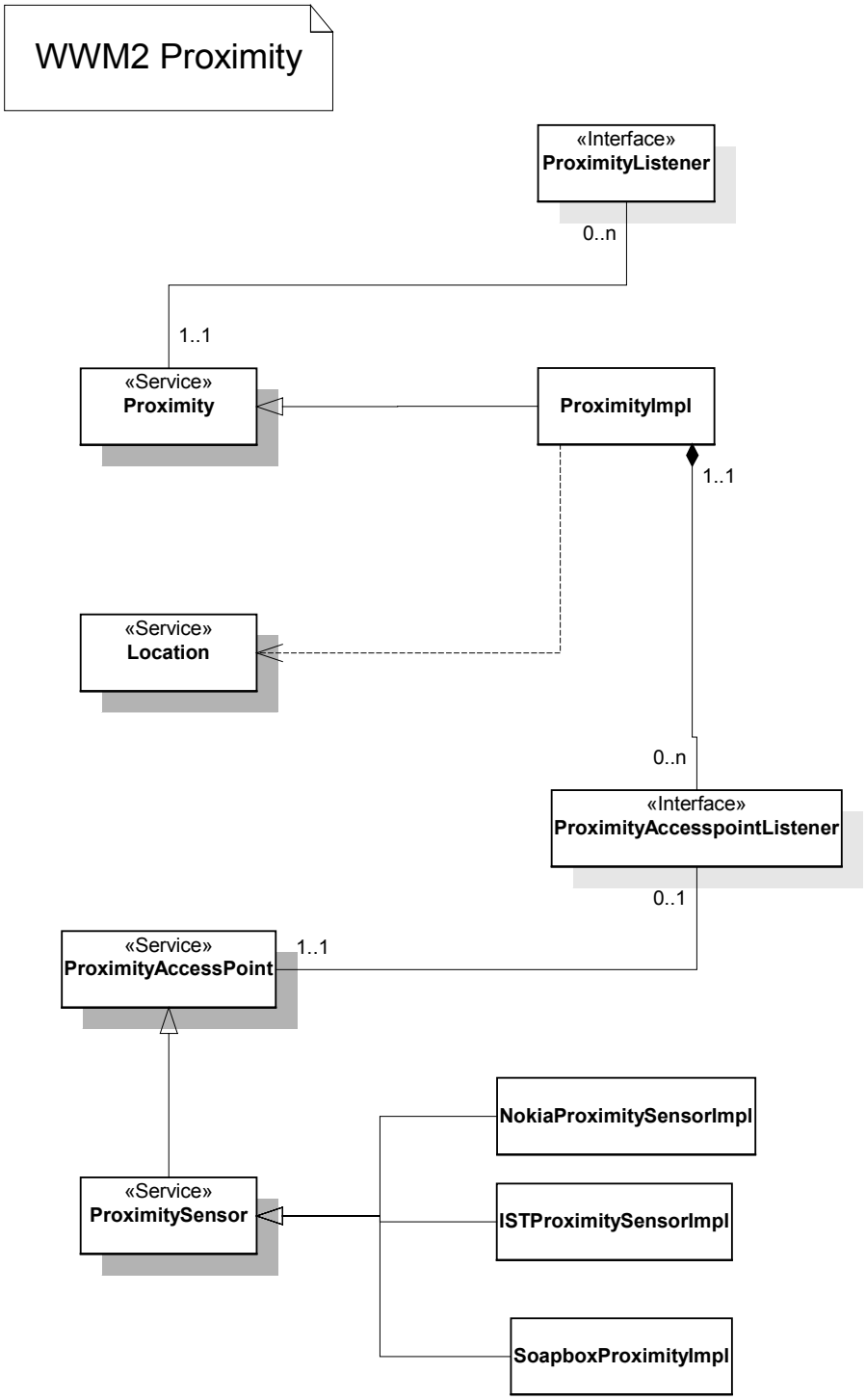

Figure 3. The Proximity bundle structure. Registered service interfaces are shown with Service stereotype. `Proximity` *users log themselves as* `ProximityListener`*s to Proximity Service. The Proximity implementation detects the ProximityAccessPoints registered to the framework and creates a ProximityAccesspointListener for each of these. The SoapBoxMaster, IST multilink unit, and Nokia Proximity Sensors may all provide ProximityAccessPoint service.*

It is assumed that each sensor type that can provide proximity information (Proximity Sensor) has an internal nominal range (in meters) to be able to detect its counterparts. This is, however, not necessarily accurate as antennas may be used, amplifications adjusted, or if there may be objects between transmitters and receivers.

The proximity package may communicate with the Location service to infer some additional rules for Proximity of services.

Some issues have deliberately been left out from the implementation:

- How each implementing package is capable of mapping its internal identifiers to the `ServiceReference`s? It is supposed that an internal Dictionary exists to map the identifiers to the PIDs of OSGi framework registered `ServiceReference`s. This Dictionary is kept in a ResourceBundle properties file
- The discovery of new `ProximityAccessPoint` Services is not handled. In the OSGi framework it is assumed that the discovery happens through listening `ServiceRegistrationEvent`s in OSGi framework.
- Due to the fact that device discovery is in the current scheme not supported, a new/unknown PDA or Pen computer carrying an `ProximityAccessPoint` (Nokia Proximity Sensor) must be handled by an ad hoc `ProximityAccessPoint` discovery: When an HTML call from a display device is issued the server searches for the `ProximityAccessPoint` by contacting a specific port of the host address retrieved from the HTML request. The discovery is facilitated by calling the Proximity interface method proximity.*suggestAccessPointAt(InetAddress host), which adds an existing* `ProximityAccessPoint` *to the OSGi environment.*

The proximity package only deals when the proximity is detected by `ProximityAccessPoint`s and inferred from `Location` structure. **Implicit proximity of e.g. by a person, who has registered himself as a user for a `Display` device is not handled by the proximity package**. If such a proximity becomes of importance, the application using the proximity service must implement the strategy by itself. For details, see the Appendix 1: The proximity paradox

5.1.2. Location

Location bundle contains the hierarchy of locations of home. The bundle has only one interface Location, which contains methods to

- add /remove sub-locations (Location inside another)
- add / remove Services to the location
- Query whether a service fixed within a location
- get all sub-locations and services contained

Each of the individual Locations are registered to the framework. An example structure may be found in Figure 4.

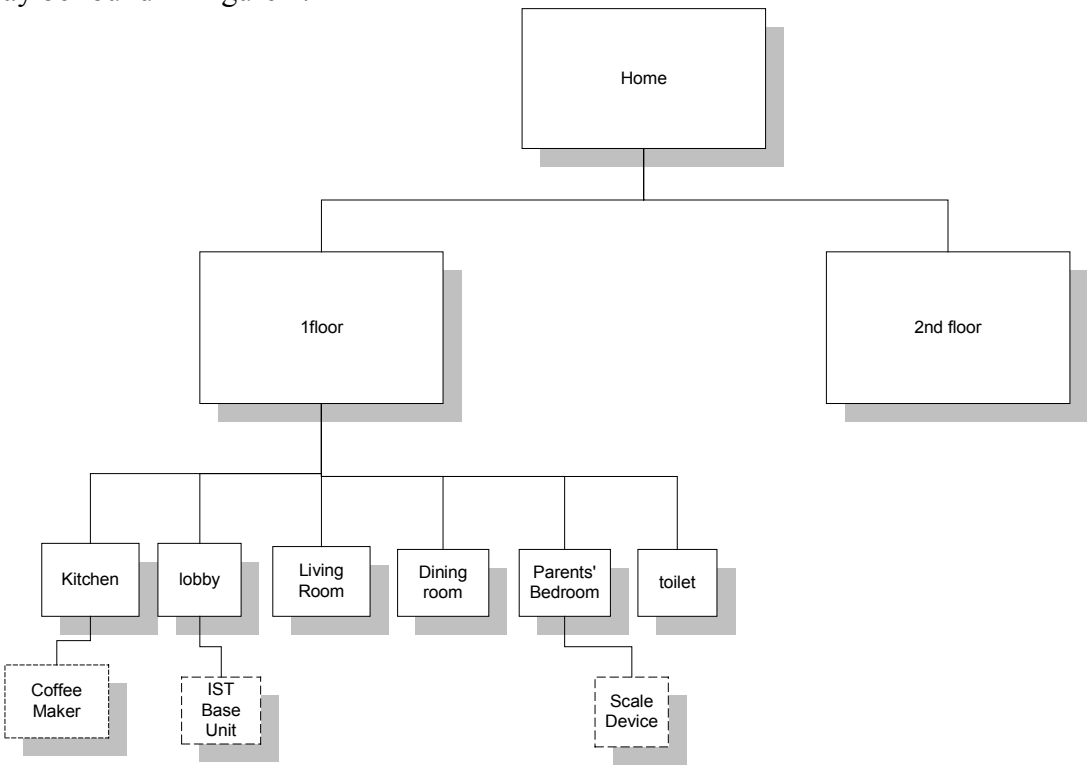

Figure 4. An example of the home hierarchy including some services.

5.1.3. Implemented Proximity Information

The implemented proximity system has knowledge about three service types: persons, displays, and WWM II UI Services. The persons and displays can have active Nokia proximity sensors (master sensors), and the displays and UI services can have passive sensors. This means, that their IDs are mapped, and known in the proximity system.

The proximity system listens to all available person services at OSGi, and connects to their Soapbox sensors, if they have been mapped to one. It also receives suggestions from the authentication, ApplicationContext, about which persons are using what displays. Those display devices are tested to see if they can provide proximity information through TCP connection.

Although the API could support quite general actions, the functionality has been restricted to support the tasks needed at the WWM2 user trials:

- Listeners to follow what is near a focus point, which can be a person or a display.
- Answer to a question, whether a certain proximity sensor ID is near a certain person.
- Answer to a question, which persons are near a specific proximity sensor ID.
- Answer to a question, whether a certain person is near a specific display.

The final implemented version of proximity information in WWM II does all required functions, but the SW architecture is simplified. The proximity related IDs should be handled only inside the proximity framework. Now the mapping of a service to its location is done by explicitly defining the proximity sensor ID for in each service that has type WWM II UI Service. All the persons and displays are explicitly given a proximity sensor ID at the ApplicationContext. Information about Nokia proximity sensors is therefore not only inside the proximity framework.

The proximity system can support also other methods than Nokia sensors, but with current implementation, it is not as easy as with the designed proximity and location framework. The main troubles are: Mapping of this other method and a service, person, or display can not be easily separated inside the proximity framework Implementation of a mid-layer, that combines the proximity knowledge of both Nokia sensors and the other method.

5.2. WWM II User Interfaces

An HTTP server included in OSGi is used to incorporate Java servlets. Each of the home appliances has one or more servlets to provide the HTML user interface. Also the home portal has a servlet. Only basic HTML code is used, e.g. no frames are supported . For push operation, the HTML code includes loading an applet.

As an extension to OSGi, all WWM II services that have user interfaces use a special UI framework. It defines interfaces for UI integrator, application, and administrative services. The main task of these interfaces is to combine the user interfaces of different services and construct a central remote controller for home.

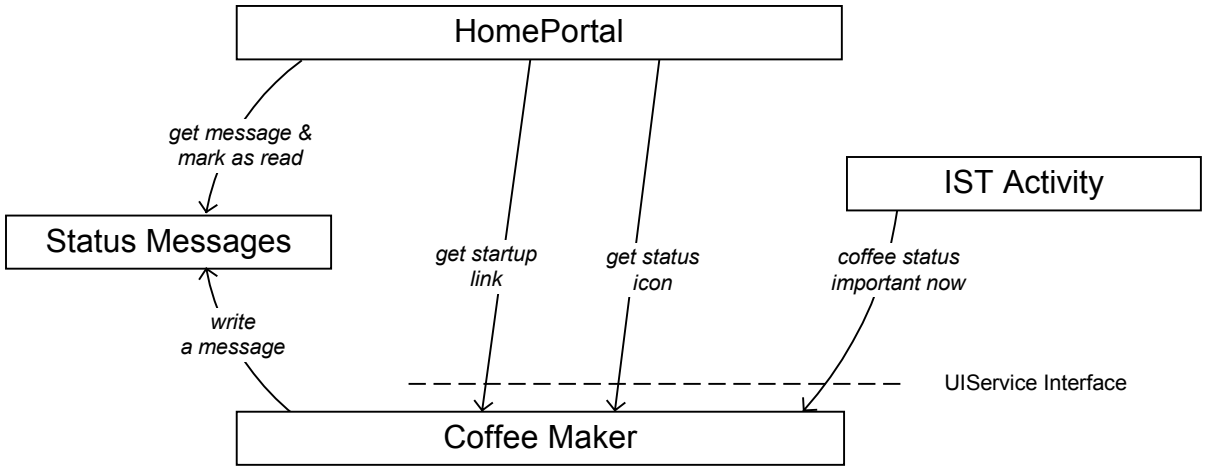

Figure 5. Operation of the UI framework. HomePortal implements UIIntegrator, and Coffee Maker implements UIService.

The UI integrator builds displays to represent the current services. Its servlet gathers the information to display from all `UIServices` registered into OSGi. The integrator builds the display by asking from the `UIservices` HTML code: their startup links, and status information.

The UI integrator provides headers and footers for the HTML code provided by application servlets. The header part presents e.g. status and title bars as suggested in [2], thus avoiding the use of frames.

UI service interface allows also simple messaging between applications. It provides a straightforward solution e.g. for a requirement "*When IST activity is low → show coffee cup"* presented in [2]. This is implemented as a method in the `UIService` interface, which indicates of an increased importance. It can be used to tell that now e.g. coffee maker status is important information.

5.2.1. Status information

Figure 6 shows how one UI intergrator, the home portal, divides the screen into sections including status information components.

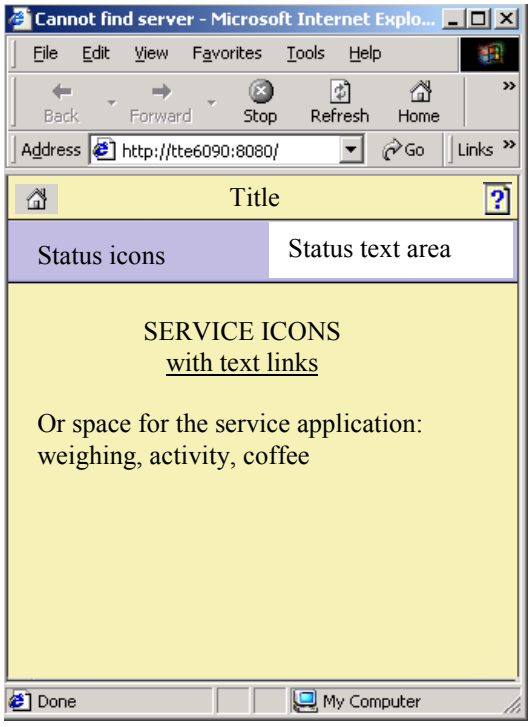

Figure 6. Sections of the display as divided by WWM II home portal.

Graphical icons are used for status information. These are context sensitive, and visible only when they are relevant. The status icons can be clicked to open the services they represent. This section is constructed, as the UI intergrator asks from UI services, whether they want to show a graphical icon.

The UI framework gives responsibility to the application services to determine what kind of status information they should show. Here are some guidelines that should be followed in applications to sustain usability at the home portal:

- Let each user configure, when they will see icons.
- Send messages only from important changes.
- It is possible to continuously show an icon to represent e.g. changing status, but then it must have real value: perhaps a good example of an icon to be shown is the activity level of a person.

5.2.2. Text Messaging

The status text messaging system of UI Integrator has an interface, which can easily be used to build e.g. a listbox. The messages are generated when an event in the house is worthwhile to be told to the user. The user is e.g. informed, e.g. that the coffee maker has been turned off automatically. For this usage, the system keeps in memory for all the messages of current day. The system supports building an UI which can separate new – unread, and old - read messages.

When an UI Service has worthwhile information it can send a text message to all the users of home, e.g. the coffee maker informs about an automated turn-off. UI integrator may listen to new messages and inform about the new message as wanted, e.g. use push with sound effects.

The messaging system has a servlet with an input line. This can be used by the operator users to type what ever text message they want to sent to the home users.

5.3. Session and context handling

`Person` and `Display` services enhance the context of WWM II services. These services are also always included into `HttpSession`. There they are directly available for all servlets that have been registered with ApplicationContext.

Display and person are often used together. To make this easy, a person service can tell all the displays that this user has logged in, and a pisplay service can tell the person using it.

5.3.1. ApplicationContext

ApplicationContext bundle provides a single authentication for all home servlets: the user has to login only once. It also provides a same HttpSession for all the servlets with the display and person objects. All this is implemented through the `handleSecurity`-method at our `HttpContext`.

Authentication

The user interfaces, mainly the `UIIntegrator`, can control authentication by indicating when the user has logged out, when the user wants to log in, and when the user wants to log in as a guest for using only public services.

When an unknown user tries to use a WWM II service, they are forwarded to an authentication servlet. It supports traditional typing of username and password, but authentication is also possible with the proximity sensor–Soapbox combination: it is successful, if a single person is detected to be in the proximity of the display.

5.3.2. Displays

Display is strongly related to `HttpSession`: they have similar lifetimes. Display concept is registered to OSGi and bound into HttpSession. The registered concept makes is possible to pose questions like: “Is there a display near person X?”

HttpContext updates the display object, so that it knows the URL that currently visible at the HTML browser. The URL defines whether or not the display is free to information push. This URL update also indicates the last time that display was used to session management of WWM2 ApplicationContext. After a decent interval, a passive session is removed i.e. auto-logout of user.

The URLs, which are a free targets for information push, are initialized by applications. It is additional information related to a servlet registration.

Home applications can use the display service to find out whether the display is public or not, and behave accordingly:

- Television user – do not allow intimate data.
- PDA user – allow intimate data.

The display service uses and controls the actual HTML push implementation by checking the intimacy and the current URL. After desired checks, it commands reload trough the push service.

5.3.3. Persons

All users are registered to OSGi as person services. The service contains only a user name and a persistent ID (service.pid). Other services do not store personal data to the actual person service but link the data to a specific person with the persistent ID. The services may use whatever data storage they want. This strategy of storing person related data is simpler than the one at OSGI v.2.0 [7]. It may be used since the requirements for personal data WWM II are simple: weighing and activity measurements are stored into SQL database, and used separately

Architecture should be further developed, if personal data would be integrated e.g. both the weight and activity data would be analyzed in a single application. The easiest approach would be not to use a complex person object, but instead a distributed storage.

Negative effect of our approach is additional data management. If person related data is stored into the Weighing bundle, that measures the data, removing a person from the system leaves obsolete data into this bundle. As a solution, all the existing persons are persistently registered to OSGi by ApplicationContext. Applications know that a person has been removed when ApplicationContext service is an active service, but there is no person service with the persistent id that application had stored to its data.

5.3.4. Configuration

Following a simple configuration model, the ApplicationContext bundle has a properties file. This file is especially important as it defines persons: the user names, persistent ids, proximity ids, and also the displays and their proximity ids.

5.4. Device discovery

Discovery mechanisms such as Jini, uPnP or MDDP developed in an earlier project [8] might be chosen. To include a device discovery protocol in to the framework would require at minimum an installation of a device discovery driver (e.g. MDDPDriver) to OSGi framework, which is first capable of detecting a device in the IP network . After being detected and registered by the appropriate device discovery mechanism, the `Device` is taken care by the `Device Manager`, which searches all `DriverLocator` services (e.g. URLLocator of MDDPDriver) to download and install a `DeviceDriver` for the installed `Device`. If such are found the `DeviceDriver.attach`(ServiceReference basicDevice) method is called, which creates and registers the new Device objects using the properties in the ServiceReference to the basic Device registered by the discovery mechanism.

WWM II architecture doesn't issue the device discovery mechanism, because that is an issue related to device driver implementation. Currently device discovery is not not implemented in WWM2, except for Weighing Scale (MDDP).

5.5. Device communication

All the WWM II devices are wrapped as IP servers, having their driver counterparts in the OSGi environment. By the driver we mean software included in to a bundle , which handles the communication between OSGi and the above device IP server. We concentrate here on the services registered by the different bundles to the OSGi framework, not on the details of IP communication between the servers and drivers.

Each of the following devices may use its internal device detection protocol. The discovered devices are registered to the framework. It is assumed that the driver has a configurable persistent storage file of devices, which can be added/removed by editing the storage directly (properties file, XML file...), or through a separate configuration panel. The persistent storage acts as a default source for creating devices when restarting. Thus – if no device discovery exists – the only way of introducing new devices is by editing this file , or by creating new devices by the management panel for the driver.

5.5.1. SoapBoxDriver

The SoapboxDriver communicates with the SoapBoxServer.

Because SoapBoxes don't have any device discovery mechanism, a properties file (which is not necessarily in Soapbox driver) contains the pre-configured information of attached SoapBoxes – either NokiaProximitySoapBoxes (SoapBoxes connected to Nokia Proximity Sensors), or MettlerScaleSoapBox (SoapBox connected to Mettler Scale).

The SoapBox driver registeres a SoapboxMaster service to OSGi. This is used also by another bundle: ProxBox. It provides special methods to get connections into those Soapboxes that are attached into Nokia proximity sensors.

5.5.2. MettlerScaleDriver

The MettlerScaleDriver is responsible of communicating with the MettlerScaleServer, directly trough TCP connection, or indirectly by using the soapbox bundle and Soapbox RF connection. The bundle registers a `Scale` interface to the system.

5.5.3. ISTMultlinkDriver

ISTMultilinkDriver communicates with the ISTMultilinkServer. It registers an `ISTDispatcher` to OSGi. The service provides IST data for interested listeners, and it can also support a short time listener for a context sensitive usage of the IST wrist unit button. Two different listener types exist, one for raw IST shout protocol data, and one with some time filtering with the activity values.

5.5.4. NokiaProximityDriver

NokiaProximityDriver communicates with active proximity sensors trough TCP and with Soapbox connection as provided by the ProxBox bundle. Currently, this bundle also registers and implements the whole Proximity Information Service, described in section 5.1.3.

5.5.5. SoehnleScaledriver

SoehnleScaledriver communicates with the SoehnleScaleServer, and creates and registers a `Scale` service for person weighing.

5.5.6. X10Driver

The X10 control module, CM11 PC powerline, is operated by an IP based server, made py J. Peterson. [10]. An electricity bundle in OSGi communicates with the server via TCP connection.

The electricity bundle provides the control mechanism of X10 devices. Because X10 has no service discovery mechanism, the bundle defines the x10 devices (LM 565 Lamp dimmers, AM12 power switches etc.) in a configuration file. It also has a servlet user interface to set up and operate all the X10 devices.

Each of the X10 devices is registered to the Framework as a separate service.

X10 lm565 dimmer

Each instance shall be registered to OSGi framework as a `LightSocket` service.

X10 lamp module

Each instance shall be registered to OSGi framework as an `EleSocket` service.

5.5.7. CelotronShopController (optional[3])

Two IP-based functionalities are used from the Celotron Shop Controller. The bundle registers service to OSGi, which provides information of temperature, and motion detector alarms..

5.5.8. The TV unit

The TV unit is not shown in Figure 1. TV unit doesn't require any specific driver software registered with OSGi.

The TV unit consists of a PC equipped with a television card and proprietary remote controller enhanced by a separate mouse device. The PC runs the web browser software. TV mode / HTML -browser mode is selected by toggling the remote controller TV/AV mode

The Web Browser software informs in login URL parameters that it is a public display device and has bigger screen size than the PDA. Otherwize, the TV unit makes a connection with the OSGi server like a normal display device. The ApplicationContext creates and registers a corresponding `DisplayDevice` object as it does with all HTML browsers connecting the system.

5.6. WWM II agents

In addition to the devices and device drivers WWM II contains some applications. They are built upon these drivers and implement UIServices in the WWM II framework. These are called agents, but also application, and service terms are used.

5.6.1. Home Portal

The role of a Home Portal agent is to introduce the `UIIntegrator` service for the rest of the bundles to use, i.e. it provides an integrated access to all available agents.

[3] The Celotron device was finally available with adequate instructions at a time, when the implementation phase of this project ended. Only some test programs for the creation of a driver exists.

5.6.2. Weighing

The weighing agent is an enhancement to the person `Scale` service. It provides an `UIService` object with a servlet for the display of weight history. The weighing bundle stores the measurement data to an SQL database, and can read from the database the the latest result of users.

The graph image is built in another bundle, and HTML code points to this another servlet with the SRC parameter of IMG tag. A graphlet bundle prints the image using an external graph library: KavaChart [11]. Data to this graph is read by a weightgraph bundle, which also knows the structure of weight data in SQL database.

5.6.3. IST Activity

IST Activity agent stores the activity measurement data internally and provides a service to look at the activity history data. It provides the `UIService` for the activity history servlet. The graph images is provided in a similar way as in Weighing.

The Activity agent listens to ISTMultlinkDriver. One listener writes plain data according to the SHOUT protocol [12] to activity database. Two other listeners use the time filtering of activity data as provided by the ISTMultlinkDriver. One handles the activity status, and another controls the electricity network of home - X10.

5.6.4. Coffee Maker

The Coffee Maker agent reads data from the coffee scale (`MettlerScale`), interprets the data and transforms it into the information about amount and freshness of coffee. The Coffee Maker agent provides an `UIService` to present this information to the user. It also uses the `Electricity` service to turn of the coffee maker power when the pot is empty or the coffee is too old.

5.6.5. Hot Spot

Hot Spot agents implement `UIService` interfaces. They define a proximity sensor identification, and the startup link of these services is an URL pointing anywhere to the internet. The Hot Spot service includes a configuration file and an user interface to define the agents.

5.6.6. Cottage Watchguard

With the Celotron driver available: This service shows a temperature of the summer cabin, and stores images from web-camera, when Celotron driver produces a movement alarm. These images are archived, and available to be browses by the users. The cottage Watchguard has `UIService` and a servlet for this. When movement is detected, the watchguard uses text messaging system to alert the users.

Without the Celetron driver: Neither the temperature sensor nor Movement detector are in use, and e.g. the movement alarm has to be input to the messaging system by an

operator user. `UIService` shows no temperature information, and the image of a summer cabin seen in the servlet is always the same picture, not web-camera.

6. Software packages

Each bundle contains one or more software packages. The documentation of software packages is in a separate document generated by Javadoc.

References

1. Luc Cluitmans, Ilkka Korhonen, Lasse Pekkarinen, Timo Tuomisto, Arto Ylisaukko-Oja, WWM II Physical Network Specification v2.4 (28.09.2001), Research Report, VTT Information Technology, 2001.

2. Lasse Pekkarinen, Katja Rentto, Mark van Gils, Timo Tuomisto, Ilkka Korhonen, WWM II User Scenarios v1.0 (28.09.2001), Research Report, VTT Information Technology, 2001.

3. OSGi Service Gateway Specification, Release 1.0, May 2000. http://www.osgi.org

4. OSGi Specification Overview, Version 1.0, January 2000. http://www.osgi.org

5. Java Embedded Server 2.0. http://www.sun.com/software/embeddedserver/

6. Java Embedded Server patch 2.0.1 http://www.sun.com/software/embeddedserver/

7. OSGi Service Gateway Specification, Release 2.0, Oct, 2001 http://www.osgi.org/resources/docs/spr2book.pdf

8. JSR 12, Java Data Objects (JDO) Specification, http://jcp.org/jsr/detail/012.jsp

9. Java Data Objects, http://access1.sun.com/jdo/

10. Java X10 CM11A/CM17A Library by J Pedersen, **http://www.jpeterson.com/rnd/**

11. KavaChart. http://www.ve.com/

12. The SHOUT protocol. A proprietary protocol by IST. Private document.

APPENDIX 1. The proximity strategy

We assume that a user registered with a display device is in the proximity of the device, or better – the device is used as his proximity information. However, this assumption is not necessarily valid: the user may not always be in the proximity of his private display device (though it is switched on). Furthermore, the user may wear another tag (proximity sensor) so that also his location is known. When a display device shows services within the proximity, is it being near the device, or near the person that counts – or a combination of both.

We have defined atomic rules for proximity service:

- PDA has always an active proximity sensor.
- If user operates a PDA without a separate proximity sensor, user & PDA location are the same i.e. user location is defined by PDA location.
- If user operates a PDA and has also a separate proximity sensor, the latter dominates, and defines the user's location.

For simplicity we may assume that in our trial system the users always have a proximity sensor-SoapBox combination, which gives the location and identity of the user.

APPENDIX 2. Thoughts on conceptual model: an intelligent agent

SW architecture functionality example around "Tired person → go and get coffee."

Our solution is simple compared to a true ambient intelligence architecture but this requirement of messaging gives ideas about that future architecture. Here is an example of how it might work:

Person context is followed in a well-defined ontology. Activity is marked low. Person context has wellness listener agents, and the one for low activity reacts. It goes checking time context information of this person. The agent finds out that the person should be active at this time and date. It starts searching refreshing possibilities from well-defined ontology at the nearby location. Since the person is currently at home, the agent examines ontology under home. There a as a property for a coffee maker it is being listed that it can offer refreshing liquid, just now ready-made in the pot. In an ambient intelligent system, the person's context would include preferences learned along time. Comparing preferences, and the refreshing possibilities, the agent might encourage the person to go and get a cup.

APPENDIX 3. Technical Limitations

JES 1.0 has never surpassed experimental status, and not been supported by Sun for a while. Especially the HTTP server of JES seems to suffer from errors. The iPAQ 3850 has network and browser related problems. For WWM2 all these arise difficulties in combination with the randomness of proximity provided by Nokia RF sensors.

JES HTTP-server

JES includes a HTTP service as defined in [3]. Although similar API exists in the newer version of OSGI [7], the functionality of HTTP service is obviously not standardized well enough. This has caused difficulties in the implementation of WWM II. This may lead into harmful dependency on the Java Embedded Server version 1.0 by Sun, and hinder upgrading into better OSGi implementations. The dependency can be understood from the section 5.3 - Session and context handling.

The version of HTTP protocol has features of both HTTP standards 1.1 and 1.0; it should rather follow only the version 1.1. The problem can be detected e.g. by starting only the original JES management panel bundles by Sun, and logging into it. You see following error messages:

```
> HttpService warning: Exception in servlet: /images/jesmp
java.net.SocketException: Connection reset by peer: socket write error
        at java.net.SocketOutputStream.socketWrite(Native Method)
        at java.net.SocketOutputStream.write(SocketOutputStream.java:83)
        at com.sun.jes.impl.http.HttpOutputStream.rawFlush(HttpOutputStream.java:242)
        at com.sun.jes.impl.http.HttpOutputStream.flush(HttpOutputStream.java:233)
        at com.sun.jes.impl.http.HttpOutputStream.close(HttpOutputStream.java:254)
        at com.sun.jes.impl.http.ResourceServlet.doGet(ResourceServlet.java:100)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:715)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:840)
        at
com.sun.jes.impl.http.ServletRegistry$Entry.service(ServletRegistry.java:140)
        at com.sun.jes.impl.http.ServletRegistry.service(ServletRegistry.java:309)
        at com.sun.jes.impl.http.HttpServer$Handler.run(HttpServer.java:458)
        at java.lang.Thread.run(Thread.java:484)
HttpService warning: Could not handle request
java.net.SocketException: Connection reset by peer: socket write error
        at java.net.SocketOutputStream.socketWrite(Native Method)
        at java.net.SocketOutputStream.write(SocketOutputStream.java:83)
        at com.sun.jes.impl.http.HttpOutputStream.rawFlush(HttpOutputStream.java:242)
        at com.sun.jes.impl.http.HttpOutputStream.flush(HttpOutputStream.java:233)
        at com.sun.jes.impl.http.HttpOutputStream.close(HttpOutputStream.java:254)
        at com.sun.jes.impl.http.HttpResponse.end(HttpResponse.java:112)
        at com.sun.jes.impl.http.HttpServer$Handler.run(HttpServer.java:480)
        at java.lang.Thread.run(Thread.java:484)
```

The HTTPSession Sharing Problem

The documentation of HTTP server in JES 2.0 claims that by using the same `HttpContext` in servlet registration calls they will share the same `ServletContext` and thus also same HttpSession. Our first implementation tried to

achieve this by registering a HttpContext to OSGi, and all servlet registrations got it trough OSGi getService method. However, the registered servlets had different ServletContexts.

The solution can be described as an additional requirement: for the HTTP server in JES 2.0: all the registration calls to HTTP server must be made inside a single bundle. Then, using the same HttpContext the servlets will have same ServletContext and HttpSession.

The implemented `ApplicationContext` bundle provides a wrapper class for the registration of servlets. UI services register their servlets through the wrapper class, and the registration calls therefore originate from a single bundle, a single service which has the same HttpContext as a member variable. As a result the HttpSession is shared between WWM2 servlets. This is a common and useful technique in servlet programming.

Resources Related System Freeze

JES can be run in a state where you can not get any data from the web server nor use the console to input. Milder version is crash of the resources part in HTTP server of JES: no images are seen on the web pages and the push applet stops working.

The total freeze occurs when all resources used in WWM2 are registered under the HTTP server of JES. System hangs when starting both the bundle *activitygraph*, and *weightgraph*.

Starting only either of the graphs, the system works with a behaviour suggesting to synchronization problems: Slow browsers like iPAQ run without errors, TV and Pen Computer run without errors, if they have to download the push applet on the first time, so that it is not yet in cache. With faster browsers downloading the push applet from JES resources shows exactly similar error message as above, with the resources being /pushresources.

The system works even after this message with a small amount of resources, e.g. without any graphs, the *istapplication* or *weighing* bundles. With more bundles present, the resources part of HTTP server freezes – it can not provide any data anymore.

Possible Causes

There may be synchronization failure inside the HTTP server of JES 1.0, which is used in its extremes, probably surpassing tests made by Sun.

Related implementation issues:

- Multiple servlets are registered to the HTTP server through a special wrapper bundle to give the same *HttpContext* for all WWM2 servlets.
- Resources are registered directly trough the HTTP server of JES, each with the simplest possible *HttpContext*.

- The graph bundles use multiple OSGi service listeners – one for each dependency. They listen for the HTTP server wrapper and use it to register their servlets.

Possible Solutions

Work around by registering all the resources except graphs to another web server. Implemented successfully except for another technical limitation at the iPAQ [see below].

Other ideas – unimplemented:

- Write a special servlet class to provide the resources instead of the HTTP server of JES. At least it would be possible to debug what is going on with the resources.
- Place also the graph servlets under another web server. The weight and activity data can be shared trough the common SQL database. URL query parameters need to be used to pass the username as a plain string. This leads into a security hazard that should be solved with some suitable protocol.
- Use GateSpace, ProSyst or at least their HTTP server to handle the resources of WWM2 services.

iPAQ 3850

Applet Start-up

To surpass the problems of the JES HTTP server, the push applet was transferred under another web server. The HTML code for starting the applet therefore defines archive location with an URL like http://localhost:8085/resources/pusharchive.jar.

The Explorer of Microsoft Pocket PC 2002 does not seem to be capable of starting the applet. With all other browsers and hosts this usage of a different web server than the one in JES makes WWM2 system fully functional.

Slowness in Push Operation

When receiving multiple push commands e.g. moving fast between two proximity sensors, the PDA can not reload the HTML pages fast enough. The iPAQ it is capable of receiving another push message only after it has fully loaded the HTML page including the push applet. This is because in the current implementation push applets are restarted continuously. Each push applet starts up, listens for one message, loads the new HTML page, and kills itself.

Possible Solutions

- Do not destroy the push applet after; implement a simple protocol for the push operation, so that loading each HTML page informs the push applet with the current browser status. Becomes somewhat complex, since push must be context-sensitive: e.g. if user is busy using another service, stepping on the scale must not change the display into weight graph screen.

- Cache and delay push messages. Implement a method for the push applet to tell the push server that now it has been fully loaded.

Network Configuration

Internet connection via WLAN and a PC as the router machine was successfully tested. However, the push worked really slowly, about 1 refresh in 1 minute, when a real Internet connection was in use.

The slowness originates in searching order of the network, as iPAQ does not seem to have any known means to configure special local addresses, instead it always does a inevitably failing and time consuming search for the local machines from the domain name servers of internet service provider.

IPAQ should have methods to configure its network e.g. by specifying the local addresses in a similar manner as using the lmhosts-files in Windows.