Dynamic Web Services on a Home Service Platform

André Bottaro[1, 2], Eric Simon[1], Stéphane Seyvoz[1] and Anne Gérodolle[1]

[1] *FRANCE TELECOM R&D*
28, chemin du vieux chêne,
38240 Meylan. FRANCE
{firstname.lastname}@orange-ftgroup.com

[2] *GRENOBLE UNIVERSITY*
Laboratoire LIG-Adele
38041 Grenoble, Cedex 9, France
{firstname.lastname}@imag.fr

Abstract

The Home Network is a pervasive environment by nature. Its openness to dynamic distributed and heterogeneous devices brings great challenges in home application design. We present here an open architecture for home service development. Distribution and heterogeneity are managed by service-oriented drivers leveraging the "service platform" concept. Our project provides an open source driver combining Web Services and OSGi standards to compose home services. Performance test results illustrate the service-oriented design.

Keywords: SOA, pervasive computing, home network, OSGi technology, Web Services, DPWS

1. Introduction

The home network is realizing the vision that Mark Weiser described years ago [1]: our environment is more and more filled with networked devices. Some of them are even slowly fading into the background as envisioned by Mark Weiser. For instance, the home DSL router will be automatically managed by servers on the Internet thanks to the work of the DSL Forum and home devices automatically configure the router thanks to plug-n-play mechanisms standardized by the UPnP™ Forum to seamlessly access the Internet from inside the home.

However, although services dynamically composing distributed device features start to appear, the challenges of Pervasive Computing remain incompletely solved. Designing an application dynamically composing the adapted device features found into an open environment according to the environment context still remains a complex task.

The Home Network emphasizes the envisioned environment openness to networked entities. First, it is open to dynamic connections: devices enter and leave the network, providing context-dependent features (e.g. according to users' activity). Second, it is open to heterogeneous devices: protocols and device types differ according to application domains and service providers. Moreover, devices are spread over the home space. Designers of innovative home applications therefore face three main challenges in this pervasive environment: dynamicity, heterogeneity, distribution.

Here, we share a platform-centric vision of the domestic network. It leverages the asymmetry between light standard devices and rich service platforms where integrated composition takes place. Our vision relies on the service and component paradigms whose marriage gives birth to the *service platform* concept.

On top of this service platform, we build an open environment that allows designers to model networked applications as a local composition of uniform service interfaces lately bound at runtime, i.e. to deal with the three identified challenges: dynamicity, distribution and heterogeneity. Devices are dynamically reified on the platform thanks to bridges dealing with device dynamic availability and distribution aspects.

The purpose of this paper is to present the design of one of these specific service-oriented drivers at the heart of our architecture. The seamless integration of local and distributed services using various protocols is one of the issues addressed by Anso[1] and Amigo[2] European projects. Jointly with Schneider Electric, we built a DPWS (Devices Profile for Web Services [2]) driver on top of an OSGi service platform [3] thanks to design patterns described in the paper. Part of this project is already published through the OSGi standardisation process [4][5].

---

[1] ANSO is partially supported by the French Ministry of Industry under the European ITEA program.

[2] This work is partially supported by the European Commission under IST Amigo Project.

The paper is organized as follows. The next section details the challenges in the design of home applications and the answers brought by novel paradigms. Then we present what home service platforms bring to home computing according to our vision in section 3. Section 4 details our approach handling device distribution and heterogeneity thanks to service-oriented drivers. Section 5 presents our DPWS Base Driver combining the Web Services and OSGi standards. Performance comparisons between the simple core Java™ implementation and our sophisticated driver are described and analyzed. Finally we conclude by confronting this work against the state of the art and pointing out major contributions.

2. Home application design

2.1. DPWS: A new home standard

Protocol heterogeneity is one of the main challenges to be faced in home application design. The main attempts targeting self-organization of the home network rely today on the use of plug-n-play protocols. The main ones on the IP network are UPnP™ recommended by DLNA, Apple Bonjour, IGRS Chinese competitor, and the new DPWS standard already included in Microsoft Windows Vista OS.

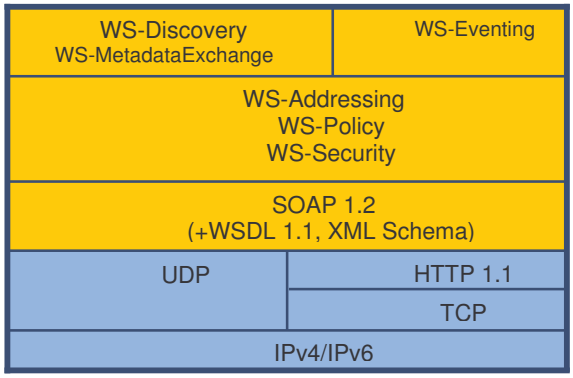

Figure 1 DPWS protocol stack

Devices Profile for Web Services (DPWS) provides a plug and play protocol middleware for IP capable devices. DPWS specifies a protocol set homogeneously based on the Web Services specifications (Figure 1). It allows devices to participate in the Web Service Oriented Architecture (WSOA) fabric.

The Web Services generality, the compliance with this well-spread Internet standard, some technical details like the DPWS Discovery Proxy that enables DPWS network to be scalable and the fact that this new standard is pushed by Microsoft are the major competitive assets of DPWS. Its generality, its XML verbosity are however potentially harmful for device implementation.

2.2. Home application requirements

Use cases attached to the smart home envisioned by telecommunication operators, service providers, manufacturers and software vendors are numerous and come along with market expectations: Enriched multimedia activities, home surveillance, and home care services for the disabled or elderly people [6].

The home network introduces hard challenges for the achievement of these use cases. These challenges are emphasized by the openness of the environment:

- Dynamicity due to device availability, device context, user location and activity.
- Distribution due to the natural location of devices in the home.
- Heterogeneity due to hardware, software and protocol variety laid by the market evolution.
- Embedded constraints due to device low cost drawn by the consumer electronics market.

In order to face these challenges, application flexibility and development simplicity are demanded. SOA paradigm (section 2.3) linked to a Component approach (section 2.4) meets some of the requirements. The technological choice of the platform achieve the flexibility needs and simplifies device availability management (section 4.1) .

Our contribution is here the design of our service-oriented drivers (section 4.2) and its implementation (part 5) relying on the concept of a “service platform” (part 3). It further answers the needs for managing distributed aspects and protocol heterogeneity in a transparent way for the developer while keeping the implied overhead below reasonable barriers.

2.3. Service Oriented Computing and local networks

The success of Service Oriented Computing is noticeable worldwide. This paradigm is mostly spread on B2B and Internet-scale innovative architectures through the use of the most popular implementation: the Web Services. The paradigm is less known in local networks whereas it brings relevant advantages in the design of home and industrial networked application.

Service Oriented Computing consists of modeling an application into logical entities providing functionalities and other entities using those functionalities. A piece of functionality – a set of

invocable operations – is called a service. The (abstract) description of a set of operations is called a service interface. A logical entity providing a service is a service provider and an entity using a service is called a service client.

The complete chart of the Service Oriented Architecture paradigm includes an entity called the service registry storing the description of available service providers (Figure 3). Service providers publish their description, which is made of a service interface and specific distinguishing properties, in the registry. Service clients actively request available providers – active discovery mode – or listens to service registration events – passive discovery mode – with the only knowledge of the needed service interface and specific properties.

The salient advantages of Service Oriented Computing are:

- Abstraction: The organization of third-party entities is considered as a service composition. Home devices are represented as black boxes.
- Loose coupling: Providers and clients only share a service interface. Clients are independent from provider implementations. Substitutability is stressed by service discovery filters that consist of a simple interface name and descriptive properties.
- Separate administration: Composed entities have distinct lifecycles.

Therefore, Service Orientation brings the abstraction needed in the composition of heterogeneous device features and the flexibility needed for the application resiliency to the home network dynamics.

2.4. Component Orientation in Service-Oriented applications

Component orientation is also needed in the Home application design. It appears to be complimentary to service orientation. The latter models the applications into a high-level service composition whereas component approaches are needed in the implementation of every identified network entity.

Salient advantages of the Component approach are:

- Code structuring: every networked entity is implemented as a logical code unit.
- Separation of the business logic and the non-functional aspects: The latter – here service distribution, service dynamicity, service (protocol) heterogeneity – have to be managed transparently for the business developer. Containers are responsible for the management of these non-functional aspects.
- Inversion of Control: The lifecycle of the component is managed outside of the component and its inner business logic. The control of this lifecycle is given to the containers.

Therefore, Component Orientation brings application modularity to every involved entity in the whole application and makes the management of complex non-functional aspects transparent to the business developer while the whole application composition is left to the administrator. As Thierry Coupaye et al. write in [7], component approaches are used in a pre-facto integration mode where all the components have to obey the same component model while service orientation is needed in a post-facto integration mode where every software entity has to be integrated just as it is with its own behavior.

3. Our vision

3.1. A platform-centric vision

We continue to believe that the self-organization of smart environments like the home network must take advantage of the asymmetry between light standard devices and rich service platforms (Figure 2). Functionalities provided by constrained devices are composed in high-level applications running on flexible platforms. Today this asymmetric vision is closed to the home reality where home servers, PCs, Media Centers, smart phones are sometimes able to control features provided by standalone media servers, media players, internet radios, lights, blinds, air condition, IP cameras. This platform-centric vision is further introduced in the previous work [8].

3.2. The service platform concept

In this vision, the application design is modular thanks to the component approach of the platform and every device feature, i.e. every service, is reified on the platform. In order to enable application components to dynamically compose those services, they are registered in a common service registry on the platform. The service registry is the masterpiece of the software design of the execution environment. This service-oriented design, which relies on the design pattern named Service Locator [9], characterizes the service platform concept used in this article.

We consider that the service platform is the smart dynamic receptacle of pervasive entities. This adaptable receptacle turns pervasive software composition into the composition of uniform service

components. The main role of the platform is to adapt to the environment in locally representing all the relevant external entities with their specific context on the platform.

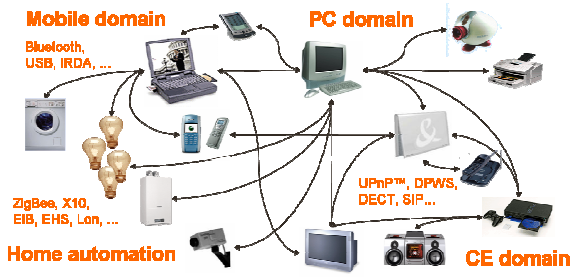

Figure 2 An asymmetric home network

4. A framework for home applications

4.1. OSGi platform: A modular platform

One of the main standard execution environments allowing application reconfiguration at runtime is the OSGi platform [3]. The core features of the OSGi platform are based on an original Java™ class loader architecture that allows for code sharing and isolation between modules called bundles, and modular software update at runtime, which are relevant platform features answering application dynamic availability in smart environments.

Moreover, the OSGi specification defines a cooperation model between bundles that is service-oriented and relies on a service registry and service eventing mechanisms. This model makes the OSGi platform enter our definition of a service platform.

The service-oriented pattern is well implemented when service interfaces (Java™ API), service requester (Java™ API client) and service provider (Java™ API implementation) are contained into separate bundles and when requester and provider cooperate through the service registry (Figure 3).

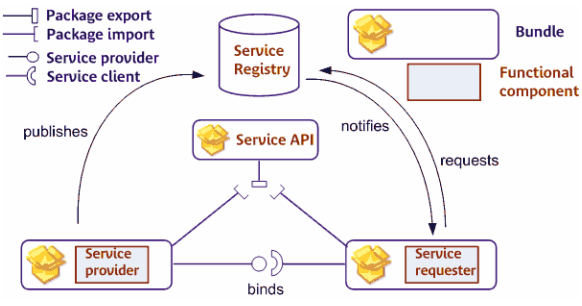

Figure 3 SOA pattern in OSGi design

Furthermore, the OSGi specification now provides a model that automates service binding between functional components. The OSGi Declaratives Services specification defines a new component model above bundles which service dependencies are automatically managed by a Service Component Runtime. This model, coming from the original project named Service Binder [10], simplifies the management of service dynamicity for developers.

4.2. Service-oriented drivers

In order to control devices on the network, we define drivers above our service platform. Every driver is designed in order to hide protocol details and distributed aspects to the developers while reifying the network dynamicity on the service platform. In this section, details of the design is explained explicitly for DPWS set of protocols. It is also applicable for other plug-n-play standard in which discovery and eventing protocols are defined.

On the one hand, every networked DPWS device is dynamically reified – imported – into a service on the platform according to programming language interfaces matching the DPWS standard description. On the other hand, every local service implementing the same programming language interfaces is dynamically reified – exported – as a DPWS Device on the local network.

The service-oriented driver characteristics are:

- Protocol heterogeneity management: This design follows the OSGi Device Access Specification: It enables the coexistence of several independent drivers on the platform without a stricter cooperation API than the OSGi general one. Thus, adherence to a pluggable model specific to distribution management is avoided. Protocol heterogeneity is simply managed by this design where every driver matches one set of protocols.
- Distribution transparency: The design completely hides protocol details since the developer only access proxy object through programming language calls. Moreover, we designed a unique API for imported DPWS services and OSGi services to be exported. Thanks to this design, client bundles seamlessly handle resident and reified DPWS devices. Controlling local and remote services is not distinguishable at development time.
- Mutualised support for (generated) refined drivers: Drivers for specific DPWS devices (printers, media servers, etc.) can be implemented above this

device API representing general DPWS devices (see refined drivers definition in [8]).

- Network dynamicity reification: The use of the OSGi service registry and its associated service eventing runtime mirrors DPWS networked discovery and eventing mechanisms, specified by WS-Discovery and WS-Eventing standards.

Thanks to this bridge, developers are able to implement applications dynamically discovering and controlling DPWS devices and services without a deep knowledge of the underlying communication protocols.

5. Our DPWS Base Driver

5.1. Modularity and requirements variety

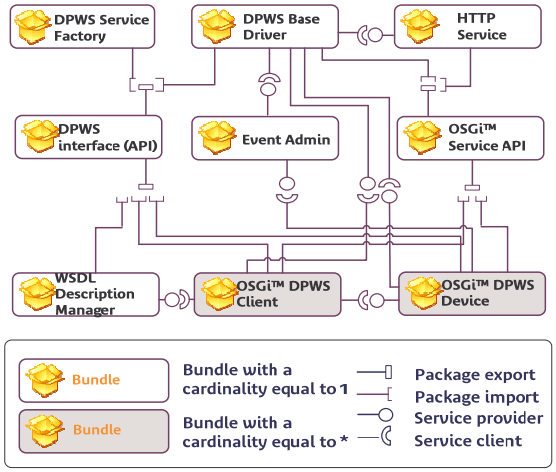

Figure 4 Global system modularity

The design of our DPWS Base Driver is modular (Figure 4) so as to face the use case variety. Thus, only a precise set of bundles is useful for every use case. Bundles can be installed on-demand thanks to the capabilities of the underlying OSGi platform.

DPWS defines the interaction between devices and clients (a.k.a. control points). Three main technical use cases are defined in the next sections: the generic network control point use case, the specific application client use case, the specific DPWS device use case.

5.1.1. Generic network control point use case

The preliminary test of the generality of our base driver is to be able to discover any device present on the network, to introspect its metadata information, invoke the provided operations, as well as to listen to device events. In this use case:

- Resource-consuming WSDL document parsing is needed in the use case. A complete generic control point has to parse this description so as to offer the detailed list of the available operations to the user.
- Export mechanisms are useless in the use case: The described generic control point only needs to access imported device references and their operations reified into objects and methods.

5.1.2. Specific application client use case

Some driver users wish to design specific client and server applications tailored to work together in a statically defined way. In this use case, two resource-consuming features are not mandatory.

- WSDL document parsing is useless in the use case: As the control points and devices are designed to work together, application designers precisely know the interactions linking them.
- Export mechanisms are useless in the use case like in the first one.

5.1.3. Specific DPWS service use case

In a constrained environment where only a simple device is to be exported, two resource-consuming features are not mandatory:

- WSDL document serving and moreover online generation can be considered optional: Constrained devices will only provide a URL for clients to find the device description.
- Import mechanisms are useless.

5.1.4. A modular design

WSDL parsing is externalized in a bundle named WSDL Description manager, as we wish that the Base Driver remains as minimal as possible. The ability to discover and access any device ensures the generality of our driver. The core driver only provides simple references to the location of WSDL descriptions. The control point is free to use the optional WSDL description manager to get an object representation.

A device manufacturer may wish to simplify DPWS integration on its platform. The optional DPWS Service offers a simplified API for the exposition of OSGi services as DPWS devices.

This modular design allows for the installation of the full set of bundles on some smart devices while a limited set will satisfy the needs of some constrained devices depending on the needs of the application.

This design also leaves the specification open to various Java™-Web Services interface description mappings. For instance, the WSDL Description

Manager may implement JWSDL standard (JSR110) mapping.

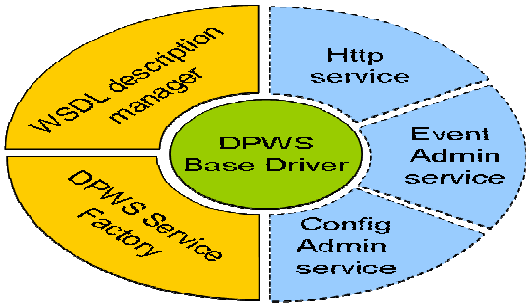

Figure 5 A modular driver

The DPWS Specification consists of several WS-* specifications, which further stresses the need for modularity. A DPWS Stack or Driver may integrate all the code parts corresponding to these specifications.

5.2. Handling asynchronous communication

The DPWS base driver applies the Whiteboard pattern to subscribe on targeted services: an internal control point registers a handler with a filter property in the service registry; the DPWS Base Driver discovers the handler and uses the filter to subscribe to matching event sources. The subscriptions can be mutualized for a set of control points matching the same targeted service with the same action filter. The base driver uses a publish/subscribe mechanism to deliver the events to the resident control points.

5.3. Lazy and Immediate Networking

The behavior of an application should be adapted to specific networked context. That is why two networking modes are defined on the driver.

On a wide network (e.g. an enterprise network), a consequent set of devices is likely to be present. For network bandwidth saving purposes, the base driver can be configured in the lazy networking mode to reify the devices with the minimum set of available information. It only retrieves device information from the network when the application asks for it.

The immediate networking mode is targeted for networks with high bandwidth and a sufficiently low number of devices. In this case, the base driver reifies the devices with all the information it is possible to retrieve at discovery time.

5.4. Lazy and Immediate Loading

For memory saving purposes, the base driver can be configured according to the needs of the application and the device constraints. The late reification minimizes used memory space. However, it slows down the first access to a targeted service.

5.5. Lessons learned

Lessons learned during the design and implementation of the DPWS Base Driver are numerous. First, the DPWS specification is a set of very general standards based on SOAP. While protocol generality can be considered as an advantage that can lead to acceptability and even interoperability, it brings also complexity in the design of a concrete framework. Understanding and implementing the set of DPWS protocols are really more complex.

Second, as Zeeb et al. describe in [11], we faced the lack of mandatory statements in the DPWS specification. Many statements are qualified with “SHOULD”, which consequently means that the statement is simply recommended and this recommendation is optional. Moreover, the novelty of the specification explains that concrete practices are not known today and some specification statements remain ambiguous. In order to remain generic, our API and our implementation have therefore to accept both behaviours in every recommendation case.

Third, it appeared that some parts of the OSGi specification and the Web Services, so-called “universal” middleware standards, do not match. The biggest issue appears for the design of the WS-Eventing mechanisms with the OSGi Event Admin ones. The WS-Eventing event filter – action URIs – can not be inserted in the OSGi Event Admin filter. We finally did not follow this OSGi chapter.

6. Evaluation

6.1. Test description

A test scenario consists of a series of 1000 invocations on a simple device action. Each test scenario is performed with a unique client calling one of the two distinct DPWS devices: a standalone DPWS device – embedding the core Java™ DPWS stack – versus an OSGi device service.

The client and the devices are installed on two distinct machines: the standalone device and the OSGi device run on the first machine whereas the client is located on the second.

- test1: the client subscribes to an event source, and invokes 1000 times an action generating an event. Invocation and event reception are synchronized.

- test2: the client calls a request-response action 1000 times.
- test3: the client calls a one-way action 1000 times.

In every scenario, an Olympic average is calculated: Every test is executed 10 times (Figure 6), the two slowest results (usually corresponding to a warm-up) and the two fastest results are not used.

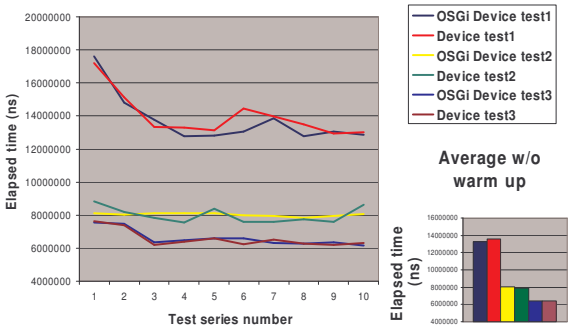

Figure 6 Performance tests

6.2. Results analysis

In the worst case, the access to OSGi devices is 1,88% slower than the direct access to the original device: 8036276ns comparatively to 7887711ns.

We performed the same tests with a OSGi control point, the results show that in the worst case, the OSGi control point is 1,606% slower than the standalone control point: 7933849ns versus 7887711ns.

The overhead brought by the service-oriented driver design is reasonable in comparison to the development advantages. The benefits naturally are the simplicity of DPWS application development and the flexibility of the overall architecture separating the business logic from the use of the given protocol. Developers thus get rid of the complex distributed aspects and overcome the protocol heterogeneity challenge.

Moreover, the best case in which the device and the client are collocated on the same platform is not described: Thanks to the driver characteristics, direct programming language calls replace network calls.

7. Related Work

Most of the projects facing distributed aspects in Component engineering rely on a direct proxy and stub generation for every component interface that has to be distributed [7][12]. In our architecture where services are numerous and heterogeneous, we advocate the reuse of several components in the import and export mechanisms that mutualize some of the distributed management aspects: Generated proxys, which are specific to explicit service profiles (Light, media servers, etc.) rely on a same base driver that relies itself on a common standard HTTP Service shared by all the applications on the platform. To have a deeper insight on our driver generation relying on the generic API of base drivers, see [8].

Marco Aiello [6] advocates the use of the Web Services at home for home device interoperability. He defines *total interoperability* as being the achievement of openness, scalability, heterogeneity. We are in line with this vision on the two first points. First, our architecture is open to dynamic connections according to standard protocols. It is also scalable since the service composition can be hierarchical or peer-to-peer (scenario 3 and 4 in [6]). However, we think that heterogeneity concerns not only device and network resources but also the home protocol variety. That is why we adopt the concept of a service platform where the service registry reifies every entity of the network. This master piece of our architecture enables the separation between the service composition logic and the distributed protocol management. The latter is even modular since every service-oriented driver like the DPWS Base Driver described in this paper is attached to one protocol and is independent to the other drivers.

This work improves our first work on a distributed OSGi platform [13]. In the latter, we mixed the service composition, precisely the service binding automation brought by the Service Binder model, from distribution transparent management here delegated to service-oriented drivers. Service binding automation and distribution transparency are independent and complementary aspects in the architecture. A distributed service registry shared by several service platform is discussed this year in the Enterprise Expert Group of the OSGi Alliance. We plan to give the details of our architecture in a next article as soon as the whole work is achieved.

This work is to be part of a wider approach [8] close to ReMMoC's [14] that aims to allow protocol adapters to be plugged in at deployment time and hide protocol heterogeneity to SOA developers. ReMMoC used WSDL-based abstractions to achieve representation uniformity: Device Profile for Web Services integration is even more relevant today. Further, our architecture shows a more loosely coupled model in which applications only adhere to the common service API of the underlying platform instead of an ad-hoc remote lookup API.

8. Conclusion

The standards of the home network make the dream of smart applications organizing distributed devices according to the user activity come true. However, the design of these smart applications remain complex. We believe that Service Oriented Computing linked with component approaches brings the needed flexibility and simplicity in targeted application design and implementation.

Our work can be distinguished from the state of the art by proposing a design for the integration of service-oriented protocols in an open modular architecture. The key concept of this design is the service platform concept that is described in the paper. The defined service-oriented drivers hide protocol details, enable a seamless control of local and remote device services, and reify network dynamicity on the platform.

The combination of the new Devices Profile for Web Services on a home standard platform is also technically new. We implemented this architecture and tested applications above the OSGi R4 service platform. Felix and Knopflerfish open source platforms are our reference platform implementations. This work was demonstrated during the review of the ITEA ANSO project in September 2007. The DPWS Base Driver is about to be delivered open source in the IST Amigo project. Actions are carried out into the OSGi standardisation process [4][5].

The main perspective of this work we are investigating is the integration of many home protocol drivers in an architecture where functions are registered in a uniform programming language API. The service composition will consist of a simplified task of chaining semantic functions together.

9. Acknowledgements

Special thanks to Sylvain Marié, Schneider Electric, who has brought his precious knowledge of the DPWS set of specifications in this work targeting a standardization proposal to the OSGi Alliance.

10. References

1 M. Weiser, “The computer for the 21st century”, Scientific American, 265(3):66-75, September 1991.

2 Microsoft Corp., “Devices Profile for Web Services”, http://schemas.xmlsoap.org/ws/2006/02/devprof, 2006.

3 OSGi Alliance, “OSGi Service Platform Core Specification Release 4”, October 2005.

4 André Bottaro, Anne Gérodolle, Sylvain Marié, Stéphane Seyvoz, Eric Simon, “RFP 86 DPWS Discovery Base Driver”, OSGi Alliance, May 2007.

5 André Bottaro, Anne Gérodolle, Sylvain Marié, "Combining OSGi technology and Web Services to realize the plug-n-play dream in the home network", OSGi Community Event, Munich, Germany, June 2007.

6 Marco Aiello, “The Role of Web Services at Home”, Advanced International Conference on Telecommunications (AICT/ICIW 2006), Guadeloupe, French Caribbean, February 2006.

7 Philippe Collet, Thierry Coupaye, Hervé Chang, Lionel Seinturier, Guillaume Dufrêne, "Components and Services: A Marriage of Reason", Technical Report I3S/RR-2007-17-FR, Mai 2007.

8 André Bottaro, Anne Gérodolle, Philippe Lalanda, "Pervasive Service Composition in the Home Network", 21st International IEEE Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 2007.

9 Martin Fowler, “Inversion of Control Containers and the Dependency Injection pattern”, http://martinfowler.com/articles/injection.html, 2004.

10 H. Cervantes, R. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model", 26th ACM International Conference on Software Engineering, Edinburgh, May 2004.

11 Elmar Zeeb, Andreas Bobek, Hendrik Bohn, Frank Golatowski, “Lessons learned from implementing the Devices Profile for Web Services”, Digital EcoSystems and Technologies Conference (DEST'07), Inaugural IEEE-IES, Cairns, Australia, February 2007.

12 Jan Newmarch, “UPnP Services and Jini Clients”, Information Systems: New Generations (ISNG 2005), Las Vegas 2005.

13 André Bottaro, Anne Gérodolle, Philippe Lalanda, "Pervasive Spontaneous Composition", First IEEE International Workshop on Service Integration in Pervasive Environments, Lyon, France, 2006.

14 Paul Grace, Gordon S. Blair, Sam Samuel, "ReMMoC, A Reflective Middleware to Support Mobile Client Interoperability", Proceedings of International Symposium on Distributed Objects and Applications, November 2003.