A Document centric Framework for Building Distributed Smart Object Systems

Fahim Kawsar and Tatsuo Nakajima
Department of Computer Science
Waseda University, Japan
{*fahim,tatsuo*}*@dcl.info.waseda.ac.jp*

Abstract

We present an architectural framework that provides the foundation for building smart object systems and uses a document centric approach utilizing a profile based artefact framework and a task based application framework. Our artefact framework represents an instrumented physical smart objects as a collection of service profiles and expresses these services in generic documents. Applications for smart objects are expressed as a collection of functional tasks (independent of the implementation) in a corresponding document. A runtime component provides the foundation for mapping these tasks to the corresponding service provider smart objects. There are three primary advantages of our approach- firstly, it allows developers to write applications in a generic way without prior knowledge of the smart objects that could be used by the applications. Secondly, smart object management (locating/accessing/etc.) issues are completely handled by the infrastructure thus application development becomes rapid and simple. Finally, the programming abstraction used in the framework allows extension of functionalities of smart objects and applications very easily. We describe an implemented prototype of our framework and show examples of its use in a real life scenario to illustrate its feasibility.

1. Introduction

One of the consequences of pervasive technologies (e.g., miniaturization of the computer technologies and proliferation of wireless internet, short-range radio connectivity, etc.) is the integration of processors and tiny sensors into everyday objects. This revolutionized our perception of computing. We are in an era, where we communicate directly with our belongings, e.g., watches, umbrella, clothes, furniture or shoes and they can also intercommunicate. These everyday objects are designed to provide supplementary services beyond the primary purpose, an initiative that has been denoted as *Smart Object* [1] computing. It has drawn significant attention from the research community, primarily because of its promising potential in various industries e.g., supply chain management, medicine, environment monitoring, entertainment, smart spaces, etc.

In this paper, we look at the system issues for building a framework for smart objects. In particular we discuss how can we build pervasive applications and smart objects with suitable infrastructure that can create a spontaneous federation to meet the dynamic and fluid nature of pervasive environment. Specifically, we focus on two issues 1) a suitable artefact framework for representing smart objects and a pervasive application model to leverage the services of smart objects dynamically and 2) an infrastructure that supports such interaction while taking care of component (smart objects in this case) management issues away from the applications. The basic idea of our framework is to use documents to externalize an application's requirements in a generic way without considering smart object management issues. Similarly, smart objects' services are externalized by structured documents. The runtime infrastructure provides a semantic association between the applications and smart objects using structural type matching of these documents. Because of such loose coupling, applications and smart objects can be built and extended orthogonally.

In the sections that follow we present the motivation and design issues of our framework. After that we discuss the proposed framework in detail. Then we present an application scenario to illustrate how the proposed framework is used. Then we discuss some generic issues before concluding the paper.

2. Motivation and Design Issues

Typically in a smart object system, context-aware pervasive applications run atop distributed smart objects embedded with awareness technologies (sensors, actuators and perception algorithms) where applications uses these objects to collect context information or to perform some services that cause changes in the real world (e.g., adjusting the air-conditioner based on sensed temperature). Considering smart everyday objects are distributed in a physical space, the characteristics and utilization pattern of a smart object system is similar to the philosophies of encapsulation and reuse behind component based frameworks, e.g., Component Object Model (COM) [7], Java Beans [5] as well as more network friendly descendants like DCOM [10], Jini/EJB

1. In this paper, smart objects and augmented artefacts carry similar meaning and will be used interchangeably.

[17], [12], OMG's CORBA [6]. Thus many architectural issues related to distributed components systems are applicable to smart object systems. In addition the highly dynamic and fluid nature of smart environments puts significant challenges for designing such framework.

2.1. Design Challenges

The characteristics of smart object systems raise a number of important architectural questions - how will we build pervasive applications to manipulate smart objects with no prior knowledge? how will we manage those unknown smart objects? how will we build application and smart objects for such dynamic environment? In the following we summarize these design challenges.

1) **Heterogeneity and Application Development:** Each of the smart objects might have different interfaces and might implement different protocols, even semantically same sm-art objects (e.g. two smart chairs from two different manufacturers) might be heterogenous from implementation point of view. It is obvious that to proliferate pervasive applications, application has to be written independently without considering which smart object from which manufacturer will be used in the application. We can not expect application to be written with prior knowledge of all of the myriad sort of smart objects of different types that it may encounter. The range of possibilities is simply too large and it is impossible to consider all smart objects during the development period.
2) **Augmentation Variation of Smart Objects:** it is hard to confine a single augmentation for a physical object. A single everyday object can provide multiple services and multiple smart objects can provide identical services with different granularities. Thus it is not possible to classify smart objects by object type. This is particularly important as this emphasizes that defining standard interfaces for smart object is not a feasible solution.
3) **Management of Smart Object:** In a conventional component framework applications are responsible for managing (locating/spawnning/etc.) the components locally, i.e., applications needs to know the access and configuration semantics. Keeping such functionalities at the level of application complicates the development process. Furthermore, for a dynamic environment like where a typical smart object system runs, it makes very difficult for application to adapt to the smart objects that change for mobility purpose or fail.
4) **Evolution of Smart Object System:** Unlike the conventional distributed component systems where applications typically resides in the digital world, smart object systems are deployed in the real world, i.e., our living spaces. An essential property of our living space is its evolutionary nature and receptibility to continual change. We incrementally organize living spaces with furniture and appliances according to our preferences and styles. Previous studies have shown how end users continuously reconfigure their homes and technologies within it to meet their demands [14]. To support the evolutionary nature of the real world it is essential that smart object systems support incremental evolution. It is necessary to have suitable programming abstraction that will allow developers to extend smart object systems' functionalities in an incremental fashion, ideally involving end users.

2.2. A Document Based Solution Framework

This paper proposes that we can address these challenges if both the smart object services and application requirements can be externalized and letting a secondary infrastructure to act as a mediator to create a spontaneous federation among them.

The challenge of heterogeneity is typically handled by existing frameworks using interface standardization [10], [6]. A programmer writes a small software to interact with a specific component / device, e.g., a networked printer. Any application can use the printer using this small software, as long as both the components (printer component, and application) agree beforehand on exactly how components will communicate with each other and the application manages this interaction locally. If the application functionlity is extended to use another device, or the same device is replaced by a new one then the application must be rewritten to interact with the new device component. Some researcher have used mobile code approach, to dynamically download the heterogenous component interfaces at application ends [17], [4]. However such approaches are impractical considering, for every new smart object an application encounters, it would need to download new codes, event for components that are semantically same. On a more lower granularity level, UPnP[2] defines a standard set of protocols for specific device types (e.g., audio/vedio devices) for interoperability. However, application that leverages these devices' services still needs to know the interfaces, and any change at the device end causes the application to fail. This is further complicated considering the nature of smart objects as mentioned in the second challenge above. It is very difficult to standardize the protocols for smart objects considering their diversity.

One way to address these issues is if we look at the functional aspects at the application end only and leave the protocol heterogeneity issues at the infrastructure end. Accordingly, we have taken a data-centric approach to handle this situation. Our framework forces an application

2. Universal Plug and Play - http://www.upnp.org

to expose its functional tasks that need the service of a smart object (i.e., a component) in a document without addressing how to access that smart object service. Similarly, a smart object is forced to expose its service features via documents. A secondary infrastructure then connects the application to the smart objects by matching the documents. However, applications and smart objects are not directly connected. Instead they communicate to the intermediary infrastructure to delegate their service requests and service responses respectively. This underlying infrastructure can provide the technical building blocks to allow applications to use arbitrary no of of smart objects as long as they provide the functionalities that are expected by the application. The infrastructure takes care of the management of smart objects away from the applications, so applications do not need to care for access, configuration or management issues. To facilitate, this communication both the application and smart objects are forced to implement a standard RESTful (HTTP/XML) communication protocol.

The last challenge mentioned above essentially asks for a decoupling among the features of both the applications and smart objects. We have addressed this challenge by following a core-cloud development model for smart objects. The core of a smart object is a generic runtime that can hosts any number of smart features as plug-in. Each smart feature is called a profile in our framework. This design allows us to decouple the smart features of a smart objects and applying same features in multiple smart objects and incrementally adding features to smart objects. Simultaneously, applications functionality can also be extended by introducing new smart objects allowing some of the application tasks to leverage their features.

Our design has been influenced by two successful approaches existing currently. First one is the internet which is an excellent example of document based system. The internet is a collection of millions of anonymously authored digital documents that are encoded in a pre defined semantics that enable heterogenous platforms to exchange these documents. The fundamental issue here is the pre negotiation of the semantics. The most widely used protocol for internet, i.e., HTTP is basically acts as the envelope for this documents and provides the negotiation semantics to both the sender and recipient (i.e., servers and client browsers and vice versa) through it headers for a flawless communication. Henceforth, structured document is the primary resource and HTTP (headers) acts as the connecting glue in the internet infrastructure. In our approach, we consider applications are the consumers and smart objects are the resources. Thus if both are expressed and amended with pre negotiated semantics using documents like HTTP headers, we can easily provide a runtime association. The second influencing approach is the commonly used shell scripting to connect arbitrary programs using the UNIX pipe facility where file handles (i.e., stdin, stdout, stderr)

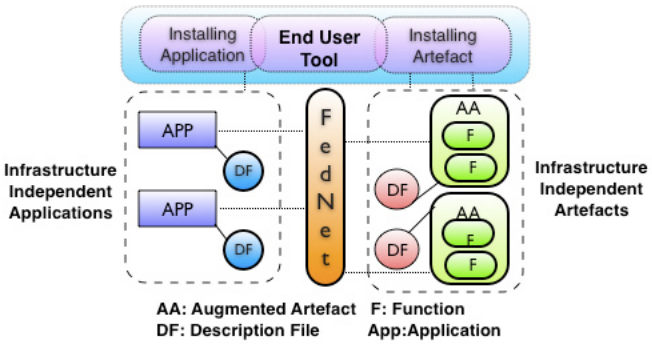

Figure 1. Basic workflow of our approach

are used to differentiate and route data. From an abstract view point we can observe that this capability of semantic mapping by pipe facility is basically the negotiation of input/output structure. Thus, a structured document with pre negotiated semantics can perform the similar piping between application and smart objects. Henceforth, documents can glue an application with smart objects given the fact that they have pre negotiation through some abstract notions. Thus our primary challenge in document based approach is to provide appropriate abstraction underneath the documents that can be utilized to build smart object systems. With these viewpoints we have designed our framework adopting following design guidelines:

1) Providing appropriate wrapper framework and abstraction with structured documents to build smart objects without concerning the target application requirement.
2) Providing appropriate abstraction with structured documents for application developers using which they can externalize application's requirements and utilize smart objects without concerning interfaces and the management of smart objects.
3) Utilizing a runtime intermediator that handles smart object management (Bootstrapping, Discovery, Utilization) and provides mapping between application and smart object services based on structural type matching thus separating the concerns of the application and the middleware.
4) Providing service extension support for both the application and the smart object using primary abstractions.

These design principles enable developers to write applications and to build smart objects in a generic way regardless of the constraints of the target environment. The basic workflow of our framework is shown in Figure 1. Our framework consists of an *Artefacrt Framework*, an *Application Development Model* and a *Runtime Intermediary Infrastructure* called FedNet. Artefact framework represents a smart object by encapsulating its augmented functionalities (e.g., proactivity of the table lamp) in one or multiple service profiles atop a runtime and allows additions of profiles

incrementally. Applications in our approach are represented as a collection of implementation independent functional tasks. These tasks are atomic actions that represent the smart objects' services. An infrastructure component FedNet, manages these applications and artefacts and maps the task specifications of the applications to the underlying artefacts' services by matching respective documents (that express the applications and the artefacts) thus externalizing smart object management and addressing heterogeneity issues away from the applications allowing developers to focus on the application functionalities only. This results in simple and rapid development of smart object systems. Primarily these two abstractions ***Profile*** and ***Task*** are used in our system and realized by corresponding documents. Additionally end user tool can be built atop our middleware independently to deploy, configure and manage the applications and the artefacts.

3. System Architecture

In this section, we present the artefact framework followed by the task centric application framework. Then, we show how FedNet utilizes these frameworks to create a spontaneous association between the artefacts and the applications. The frameworks and FedNet are implemented in Java.

3.1. Artefact Framework

Artefact framework encapsulates a smart objects so that it can be connected and used by other smart objects and applications. It provides a layered architecture where basic smart object functionalities are combined in a generic core component that act as the runtime. Additional augmented features can be added as plug-ins into the core. Each augmented feature is called a *Profile* in our approach. These profiles are artefact independent and represent a generic service implementing service specific protocols., For example: sensing room temperature could be one profile, and multiple artefacts (e.g., a window, an air-conditioner, etc.) can be augmented with a thermometer for supporting this profile. A smart object and its service profiles are externalized using structured documents expressed in XML. This document specifies the profile detail, i.e., input/output data type, methods, parameters, etc that allows data exchange, discovery, and application interaction.

3.1.1. Internal Architecture of Artefact Framework. The internal architecture of the artefact framework is shown in Figure 2 and consists of the following:

1) **Core Component:** Typically instrumented artefacts have some common characteristics e.g., capable of communication [2], [16], provides perceptual feedback, possesses memory etc. The core component of the artefact framework encapsulates all these functionalities. The communication module facilitates communication support and encapsulates the transport layer where as the discovery module allows service advertisement. The notification module enables the rest of the modules to indicate their status. The artefact memory contains property data, profile descriptions, and other temporal data. The client handler is the request broker for services and delegates the external requests to specific profiles. Finally, the profile repository hosts the array of profiles. The profile repository has dynamic class loaders to load the profiles dynamically when requested. The entire core is packaged in an executable binary and runs independently.
2) **Profile:** Each profile represents a specific functionality and implements the underlying logic of the functions, e.g., providing context by analyzing the attached sensors' data (e.g., room temperature) or actuating an action by changing the artefacts' state (e.g., increasing the lamp brightness etc.). Each profile is of type sensor or actuator and has a profile handler, a template to plug-in device code and context calculation or service actuation logic. The profile handler has an abstraction layer that hides the heterogeneity of the underlying devices.

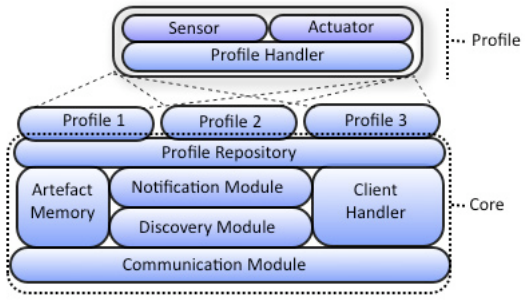

Figure 2. Architecture of Artefact Framework

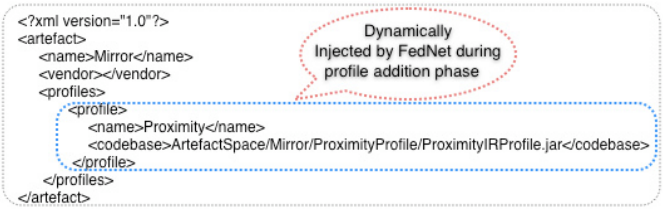

Figure 3. Artefact Description File for a Mirror with Proximity Profile

3.1.2. Documents to represent Artefacts. The artefact framework's core is packaged as a ready-to-run binary with a description document called Artefact Description File (ADF) as shown in Figure 3 for a mirror artefact with a *Proximity*

Profile[3]. Profiles are packaged as plug-ins with a Profile Description File (PDF) (Figure 6) that run atop the core. The PDF specifies the data semantics of the corresponding profile and contains a *detector* or an *actuator* node based on the profile type. The sensor profile's description follows the specification of the Sensor Modeling Language (SensorML) [13] (Figure 4(a)). The primary strengths of SensorML are its soft typed attribute, reference frame and parameters, with which the semantics of different sensor data platforms can easily be understood and interchanged. For an actuator profile[4], our custom designed XML based *Artefact Control Language (ACL)* is used (Figure 4(b)) where the *state* attribute is used to abstract the operational states of the artefacts. It contains the input parameters to change the states along with their data type. PDF also contains a quality of service(QoS) block which specifies profile's quality. Furthermore, these files contain an *installation-instruction* block that provides hardware installation guidelines.

3.2. Task Centric Application Model

An application is expressed as a collection of functional tasks independent of the implementation. This specification allows FedNet to map these tasks to respective service provider artefacts. An application developer can follow any library and implementation language to code the execution logic. The two things necessary to work in a FedNet environment are the task specification, and the generic access mechanism.

Any application is composed of several functional tasks, i.e., atomic actions. In ubicomp applications, these atomic actions may be: *"turn the air-conditioner on", "sense the proximity of an object"* etc. An application is expressed as a collection of such functional tasks in a Task Description File (TDF). Each task specifies the respective profiles it needs to accomplish its goal. Figure 5 shows part of the task description file for a smart display application explained in section 4 . Each task contains Quality of Service (QoS) requirements for the target profiles.

The second requirement for an application is to use generic web protocols to access artefact services. We consider defining strict interfaces for applications limits the portability and adoption of applications. Since any application only needs to manipulate data to interact with underlying smart objects, a compatible and consistent message is enough to enable applications interaction with smart objects. Consequently, we have addressed this concern by allowing FedNet, the intermediary component of our middleware to act as the gateway of smart objects services and accessing those services from application in a RESTful manner using

3. Proximity Profile's sole purpose is to recognize the presence of an object in front of the artefact.

4. Please note that the protocol to handle the underlying device is implemented in the profile implementation.

```
<profile>
        <name>Proximity</name>
        <purpose>Sensing the proximity </purpose>
        <type>Sensor</type>
        <detector>
                <identification>IR Sensor</identification>
                <referenceFrame/>
                <inputs/>
                <outputs>
                    <output>
                        <name>position</name>
                        <datatype>string</datatype>
                        <value/>
                    </output>
                </outputs>
                <parameters>
                    <parameter>
                        <name>timestamp</name>
                        <datatype>long</datatype>
                        <value/>
                    </parameter>
                </parameters>
        </detector>
        <QoS-attribute>
                <qos> .....</qos>
        </QoS-attribute>
        <installation-instruction>.....</installation-instruction>

</profile>
```

(a)

```
    <actuator>
        <identification> ... </identification>
        <states>
            <state>
                <name> ... </name>
                <input>
                    <name> ... </name>
                    <parameter>
                        <MIMEdatatype> ....</MIMEdatatype>
                        <value> </value>..
                    </parameter>
                    ----- More Parameter ----
                </input>
                <output>
                    <name> ... </name>
                </output>
            </state>
        </states>
        <installation-instruction>.....</installation-instruction>
    </actuator>
```

(b)

Figure 4. (a)Profile Description File for Proximity Profile, SensorML is used in the detector node. (b) Artefact Control Language is used for actuator profile

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
        <name>Smart Display</name>
        <purpose>....</purpose>
        <binaryPath>ApplicationSpace/SmartDisplay/SmartDisplayApp.jar</binaryPath>
        <accesspointIP>10.0.1.3</accesspointIP>
        <accesspointPort>9824</accesspointPort>
<task-list>
        <task>
                <id>T1</id>
                <purpose>Measuring Proximity</purpose>
                <required-profile-type>Sensor</required-profile-type>
                <profile-name>Proximity</profile-name>
                <communication-mode> asynchronous</communication-mode>
                <profile-QoS-attribute>
                        <qos>.....</qos>
                </profile-QoS-attribute>
        </task>
          ---------- More Tasks---------
 </task-list>
</application>
```

Dynamically Injected by FedNet during application installation phase to attach the Access Point

Figure 5. Task Description File (partly) for a smart display application

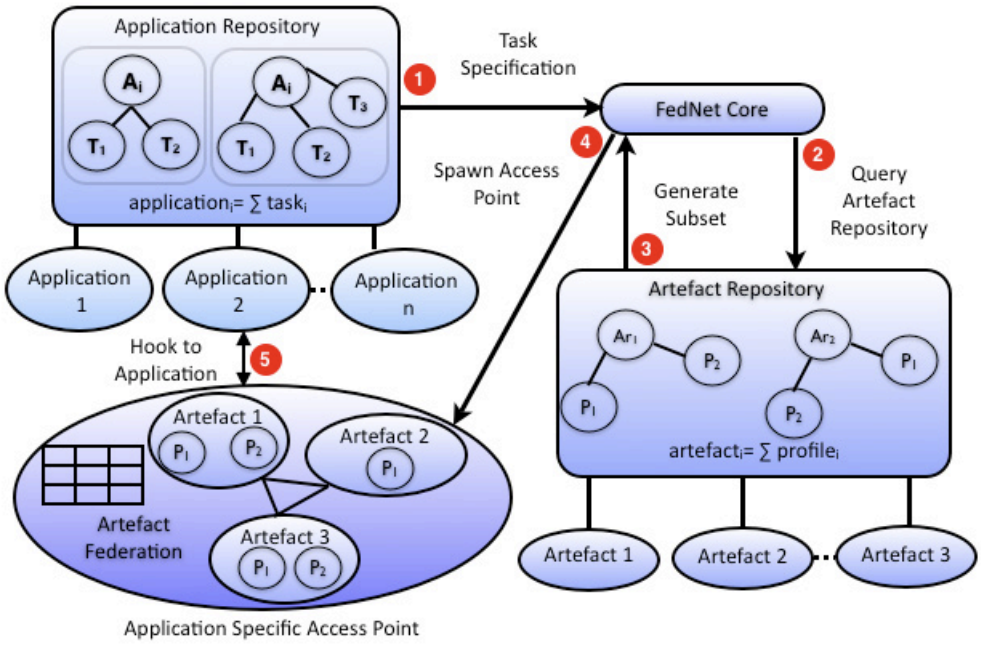

Figure 6. Architecture of FedNet

simple HTTP get and post with XML messages. During application installation in FedNet, an access point is assigned to the application. An application needs to access this point to send requests and receive responses from the underlying artefacts. During the application's instantiation time, the required physical artefacts data semantics (*detector* and *actuator* nodes of the Profile Description File) are send to the application by FedNet, to let the application prepare for the moveable data accordingly. Thus applications do not need to adhere any middleware specific interfaces to interact with the smart objects yet can leverage their services via FedNet.

3.3. FedNet Runtime Infrastructure

In our approach both the applications and artefacts are infrastructure independent and expressed in high level descriptive documents (i.e., task and profile specifications). FedNet provides the runtime association among them by utilizing only the documents of these applications and artefacts. It can contact the the artefact core using the semantics described in the artefact documents for mapping application tasks, similarly application can contact FedNet in a RESTful manner. FedNet itself is packaged in a generic binary and composed of four components, Figure 6 shows the interaction among four components.

1) **Application Repository** hosts all the applications that run on FedNet. During an application's deployment, the binary executable and the Task Description File (TDF) are submitted to this repository. FedNet Core generates the an access point for the application and updates the respective TDF by dynamically injecting the identity of the corresponding access point as shown in Figure 5.
2) **Artefact Repository** manages all the artefacts running in FedNet environment. During artefacts' deployment, the executable binary implementing the artefact framework and the Artefact Description File (ADF) are submitted to this repository. When a profile is added to an artefact, the profile information is dynamically injected into ADF as shown in Figure 3 and the respective profile is attached to the artefact.
3) **FedNet Core** provides the foundation for the runtime federation. When an application is deployed the task specification is extracted from the application repository by the FedNet Core. It analyzes the task list by querying the artefact repository and generates an appropriate template of the federation and attaches it into a generic access point component for that application. When an application is launched, the access point is instantiated and the respective template is filled by the actual artefact available in the environment right at that moment thus forming a spontaneous federation.
4) **Access Point** represents the physical environment needed by an application. Since each application's artefact requirement is different and each application might not be running all the time, FedNet assigns a unique access point for each application; meaning multiple federations of artefacts can co-exist in the

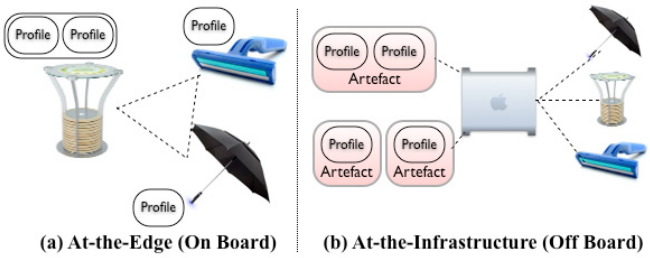

Figure 7. Location Modalities of Artefact Framework

environment. Simultaneously, each artefact can participate in multiple federations. When an application is launched, the access point sends the federated artefacts data semantics, i.e., SensorML and ACL to the application. This allows an application to know the semantics of movable data in advance. From then on, the application delegates all its requests to the access point which in turn forwards them to the specific artefact. The artefacts' responses to these requests by providing their profile outputs either by pushing the environment state (actuation) or pulling the environment states (sensing) back to the access point that are fed to the application.

3.4. Distributed Management

In the earlier part of this section we have provided the explanation of the functional roles of the primary components of our infrastructure. From physical implementation point of view all these components could be distributed, i.e., instrumented artefacts can run in their own nodes, applications can run on the artefact nodes, or in a separate node integrating multiple artefact nodes, and FedNet can run in its own node to manage all other nodes.The artefact framework essentially is the digital identity of an artefact. So an obvious issue is the location of this digital part. We have two choices as shown in Figure 7: a) At-the-Edge (On-Board) b) At-the-Infrastructure (Off-Board). At-the-Edge means the artefact itself has a processing unit that hosts its digital representation where as the At-the-Infrastructure means a proxy, running in a separate location represents the artefacts and communicates with the artefact to retrieve sensor data or to actuate artefact's function using some communication protocol, e.g., Bluetooth, IEEE 802.11x, etc. Both choices have pros and cons. While at-the-edge approach provides pre-configurable and self sustainable artefacts, it is prone to limited capability. On the other hand, although at-the-infrastructure approach requires manual configuration and maintenance, the primary advantage is the rapid prototyping support. In our current implementation we have adopted At-the-Infrastructure approach and each artefacts digital representation, i.e., artefact framework's binary core and profile plug-ins are deployed in a node that communicates with the physical artefact thr-ough some communication channel to retrieve the actual profile service via the hardware attached into the artefact. The same is true for the applications, i.e., the applications running on a single artefact can reside in the same node that represents the artefact and the application that integrates multiple artefact can reside on the any of those artefacts node. It is the FedNet components that organize these nodes in a distributed manner and manages the spontaneous federation. The FedNet components (i.e., Application Repository, Artefact Repository and FedNet Core) can reside in one or multiple nodes and manage the underlying artefacts and applications.

4. An Application Scenario

We have built several smart object systems using our middleware. In this section we are presenting one of those systems.

We constructed a smart mirror by augmenting a regular laptop with acrylic magic mirror (Figure 8(a)). Initially this mirror has a display profile. We wrote an application for this mirror where the application can show some personalized information (e.g. weather, stock quote, movie listing etc.) into the mirror display. However, this application can proactively show information only when someone is in-fornt of the mirror. But for such proactivity it requires a proximity profile. To enable this application feature, later we have added a proximity profile to this mirror by attaching an Infra-red sensor. This improved the applications interactivity. Afterwards we built a completely separate application for the mirror where user's dental hygiene is reported in a persuasive way utilizing the metaphor of a clean and dirty aquarium. We replaced the previous application running on the mirror with the new one. This new application requires a smart toothbrush that can detect its state-of-use (Figure 8(b)). We constructed the smart toothbrush by attaching a wireless accelerometer sensor and deployed it in our environment with corresponding profile. Thus the mirror shows an aquarium reflecting users brushing practice whenever the user brushed his teeth in front of the mirror. All the smart objects and applications were deployed and configured using our end user deployment tool running atop FedNet. We have also performed an informal user study for evaluating the usability of our approach form end users point of view that we have reported at other forum [9].

Both the applications were built independently and deployed with corresponding documents expressing the tasks, similarly the two smart objects e.g., the toothbrush and the mirror were built independently and deployed with corresponding documents. FedNet provided the runtime association among them thus freeing application from smart object management. Furthermore, this scenario highlighted the service extension feature of our middleware. We have added new profiles to an existing smart mirror allowing an

Figure 8. Applications' Components and End Users' Interaction

existing application to leverage new functionalities. Importantly, the application did not have to take into account the heterogeneity issues introduced by the addition of an Infra-red sensor as it was handled by the proximity profile implementation.

5. Discussion

There are primarily two abstractions underneath the documents that we have utilized in our framework. From the smart objects' perspective it is the notion of profile that handles the service implementation detail and protocol issues. Since profiles are independently built following a plugin architecture, a smart object service can be extended anytime by adding new sensors or actuators and attaching corresponding profile into the smart objects core. Also, if a specific service needs to be updated only the corresponding profile need to be replaced, not the entire smart object or the applications utilizing them. Furthermore, a profile may provide services in various granularities thus supporting multiple applications requiring services at different scale (i.e., some applications may ignore some service features). The profile notion has the potentially serious implication that standard common vocabularies or ontologies will be needed to support general interoperability of profiles and applications. However, by profile abstraction, we are not trying to define the ontology for profiles. Instead we are providing a structure that designers can use to disseminate their implemented ontology and glue it with rest of the infrastructure.

The second abstraction is from applications' perspective, i.e., tasks that simply externalize an applications requirements, so any application can be expressed with this abstraction. Not necessarily all tasks of an application can be supported by an existing environments, however with the incremental addition of new smart objects in the environment or porting application to another environment with richer smart objects might enable the full functional features of an application. In addition an applications functionalities can be updated independently (application binary and the document) without concerning the impact of such update in the middleware or smart objects. In our approach, such flexibilities are provided elegantly by only expressing applications' task specifications in documents and ignoring smart object management issues at the application level. FedNet provides the appropriate mapping of these documents with smart objects documents expressing their services.

This disassociation of applications from the smart objects they reference is identical to the Model-View-Controller (MVC) architecture from Smalltalk. In the MVC architecture, data (the model) is separated from the presentation of the data (the view) and events that manipulate the data (the controller). Similarly, documents in our middleware act as the glue that associates smart objects services to applications that manipulate the services. Such separation of concern (i.e., both the applications are artefacts are independent of FedNet and come as ready-to-run binary), and data centric approach also enable us to provide additional services orthogonally in our system. For example, we have implemented several end user tools atop FedNet that enable end users to deploy, configure and manage the applications and smart objects running in the FedNet environment.

6. Related Work

To date several methods have been proposed to address system support for ubicomp applications. One approach is interface and protocol standardization as attempted by Jini[5] and UPnP[6] respectively. Jini describes devices using interface description and language APIs allowing applications to utilize those interfaces where as UPnP attempts to standardize protocols to allow devices to intercommunicate seamlessly. These infrastructures provides well defined interfaces for application developers, however it is hard to build application integrating appliances that do not follows their specific protocols. Furthermore, these systems provide little support for extending applications or appliance services. For example, it is hard to add features in an existing artefact and using that feature immediately in the application with these infrastructures. Patch Panel [1] is a programming tool that provides a generic set of mechanisms for translating incoming events to outgoing events using EventHeap [8] communication platform. It allows new applications to

5. Jini - http://www.sun.com/software/jini
6. Universal Plug and Play - http://www.upnp.org

leverage the services of existing components. Our overall approach is close to Patch Panel as we seek to support incremental integration. However, we exploit a distributed state model with an artefact framework that enable incremental addition of features to both artefacts and applications. In SpeakEasy [4] mobile codes (typed data streams and services) are exchanged among heterogeneous devices to create an interoperable environment. SpeakEasy does not consider the incremental extension of artefact services or end user deployment as its primary focus is on service composition. InterPlay [11] is a home A/V device composition middleware and uses pseudo sentences to capture user intent, which is converted into a higher level description of user tasks. These tasks are mapped to underlying devices that are expressed using device description. Although our approach is very close to InterPlay as we employ similar mapping of tasks to device services, our challenge is to provide generic abstractions and to support incremental extension and deployment of both artefacts and applications. Our artefact framework is a major leap from InterPlay which signifies our contribution. A range of middlewares for pervasive systems [15], [3] specify their application development processes strictly. These middlewares usually provide end-to-end support for the application developer, i.e., instrumented artefacts are encapsulated into wrappers and an array of APIs is provided to the applications to manipulate them. The problem of this approach is that the applications and the instrumented artefacts become virtually incompatible in other environments. We have adopted a document centric approach allowing development of infrastructure independent applications and artefacts and the runtime association between them is provided by FedNet.

7. Conclusion

In this paper we have presented a document based approach to build smart object systems. Applications' requirements and smart objects' services are externalized using structured documents utilizing *Task* and *Profile* abstractions respectively. A runtime framework FedNet provides the dynamic association by structural type matching. The contributions of our approach are two-fold: firstly, it allows developers to write applications in a generic way regardless of the constraints of the target environment utilizing the abstractions that are realized through documents Secondly, it allows extension of functionalities of smart objects and applications very easily. We have described an implemented prototype of our approach with an application scenario that highlight the power and flexibility of our framework. We consider our approach is very useful for the ubiquitous computing domain, particularly one that involves smart objects.

References

[1] R. Ballagas, A. Szybalski, and A. Fox. Patch panel: Enabling control-flow interoperability in ubicomp environments. In *PerCom 2004*, 2004.

[2] M. Beigl, H. W. Gellersen, and A. Schmidt. Media cups: Experience with design and use of computer augmented everyday objects. *Computer Networks, special Issue on Pervasive Computing*, 35-4:401–409, 2001.

[3] A. K. Dey, G. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.

[4] W. K. Edwards, M. Newman, J. Sedivy, T. Smith, and S. Izadi. Challenge: recombinant computing and the speakeasy approach. In *th ACM MobiCom*, 2002.

[5] R. Englander. *Developing Java Beans*. O'Reilly and Associates, 1997.

[6] O. M. Group. Common object request broker architecture.

[7] D. Iseminger. *Com+ Developer's Reference*. Microsoft Press, 2000.

[8] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1-2, 2002.

[9] F. Kawsar, K. Fujinami, and T. Nakajima. Deploy spontaneously: Supporting end-users in building and enhancing a smart home. In *The Tenth International Conference on Ubiquitous Computing (Ubicomp 2008)*, pages 282–292, 2008.

[10] D. Krieger and R. Adler. The emergence of distributed component platforms. *IEEE Computer Magazine*, pages 43–53, March, 1998.

[11] A. Messer, A. Kunjithapatham, M. Sheshagiri, H. Song, P. Kumar, P. Nguyen, and K. H. Yi. Interplay: A middleware for seamless device integration and task orchestration in a networked home. In *IEEE PerCom 2006*.

[12] S. Microsystems. Enterprise java beans.

[13] O. G. C. Inc. *Sensor Model Language (SensorML) implementation specification*.

[14] T. Rodden and S. Benford. The evolution of buildings and implications for the design of ubiquitous domestic environments. In *ACM CHI 2003*, 2003.

[15] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83, 2002.

[16] M. Strohbach, H. W. Gellersen, G. Kortuem, and C. Kray. Cooperative artefacts: Assessing real world situations with embedded technology. In *Sixth International Conference on Ubiquitous Computing (Ubicomp 2004)*, pages 250–267, 2004.

[17] J. Waldo. The jini architecture for network-centric computing. *Communication of the ACM*, pages 76–82, July, 1999.