Middleware for Building Ubiquitous Computing Applications Using Distributed Objects

Nicolas Drosos, Eleni Christopoulou, and Achilles Kameas

Research Academic Computer Technology Institute, Research Unit 3, Design of Ambient, Intelligent Systems Group, N. Kazantzaki str., Rio Campus, Patras, Greece
{ndrossos, hristope, kameas}@cti.gr

Abstract. Ubiquitous systems are characterized by multi-fold complexity, stemming mainly from the vast number of possible interactions between many heterogeneous objects and services. Devices ranging from simple everyday objects populated with sensing, actuating and communication capabilities to complex computer systems, mobile or not, are treated as reusable "components" of a dynamically changing physical/digital environment. As even an individual object with limited functionality, may present advanced behavior when grouped with others, our aim is to look at how collections of such distributed objects can collaborate and provide functionality that exceeds the sum of their parts. This paper presents GAS-OS, a middleware that supports building, configuring and reconfiguring ubiquitous computing applications using distributed objects.

1 Introduction

The vision of Ambient Intelligence (AmI) implies a seamless environment of computing, advanced networking technology and specific interfaces [5]. In one of its possible implementations, technology becomes embedded in everyday objects such as furniture, clothes, vehicles, roads and smart materials, and people are provided with the tools and the processes that are necessary in order to achieve relaxing interactions with this environment. The AmI environment can be considered to host several Ubiquitous Computing (UbiComp) applications, which make use of the infrastructure provided by the environment and the services provided by the AmI objects therein. The target of this paper is to present the GAS-OS, a middleware that we developed and supports the composition of UbiComp applications from AmI objects. GAS-OS runs on every AmI object and collectively serves as a distributed component framework. Moreover, it provides developers of UbiComp applications with a uniform programming model that hides the heterogeneity of the underlying networks, hardware, operating systems and programming languages.

The structure of the paper is as follows. Section 2 outlines the design challenges of UbiComp applications and the requirements for a middleware that supports such applications. Section 3 describes the architecture of the GAS-OS, followed by a real life application example in section 4. Section 5 presents related approaches and section 6 lessons learned from this work. Finally we conclude in section 7.

P. Bozanis and E.N. Houstis (Eds.): PCI 2005, LNCS 3746, pp. 256 – 266, 2005.

2 Design Goals

The design goals of a middleware that supports the composition of UbiComp applications are tightly interrelated to the challenges arising from the deployment of such applications and also emerge from the requirements of generic middleware systems.

According to the AmI vision people will build "ecologies" (UbiComp applications) by configuring and using AmI objects; everyday objects augmented with Information and Communication Technology (ICT) components. AmI artefacts can be seen as distributed objects since data, behavior and services encapsulated must be accessed remotely and transparently to the overall application. An important aspect though is that these augmented objects still maintain their primary role and *autonomous* nature. Furthermore, in order to *compose* UbiComp applications using artefacts, they must provide the means to be easily used as building blocks of large and complex systems.

In addition, because of the *heterogeneity* of these objects, a key challenge that arises is the feasibility of *semantic interoperability* among them. As AmI artefacts are resource constraint devices, grouping them together could emerge more advanced behaviors. Thus their *composeability* is a challenge that can give rise to new collective functionalities. As artefacts in UbiComp applications may offer various services, the challenge of a semantic representation of services and a *semantic service discovery* mechanism is evident. UbiComp applications also need to be *adaptive* to changes, *robust* and *fault-tolerant* as they are usually created in an ad-hoc manner, and AmI objects are liable to failures. They must also be *context-aware* to understand the environment and adapt their behavior to different situations. *Scalability* is also very important since UbiComp applications are usually composed by a large number of objects.

Considering the users' perspective, a key challenge is the *ease of use, development and deployment*. The combination of objects needs to be based on a user-oriented and user-friendly model. This implies that objects' capabilities must be "advertised" to users through a comprehensible "vocabulary".

Considering the system's perspective, the heterogeneity of artefacts implies middleware systems on top of which applications can function transparently based on the infrastructure. To preserve the autonomy of artefacts and to cater for the dynamic nature of such applications ad-hoc networking has to be supported. The underlying physical networks used are heterogeneous ranging from infrared communication over radio links to wired connections. Since every node serves both as client and server (devices provide/ request services), the required communication can be seen as *p2p*.

Due to the dynamicity of UbiComp applications and the mobility of artefacts, the middleware has to consider services and capabilities of changing availability. Even a service that is both functional and reachable can become unavailable (volatility problem). Furthermore, as any object can become an artefact, regardless of its physical (e.g. power) or computational properties (e.g. memory), the core functionality should be small enough to be executed on resource constrained devices, but *extensible* to use the capabilities of more powerful devices as well. Therefore, we should not pose severe restrictions, like the assumption of a specific platform (*platform independence*). Various manufacturers should be able to implement their consumer solutions on a variety of platforms, not predefined in advance. At the same time, the middleware has to cope with the unavoidable heterogeneity in service definition and deployment.

Since UbiComp applications can be synthesized by end-users, the middleware has to support a user-oriented *conceptual model*. Additionally interacting with the system has to be done in *real time* since services must be available to users at each particular moment. Finally, middleware systems aiming at a broad range of applications, should remain *open*, capable of collaborating with established technological solutions and standards for communication, interoperability etc.

3 Designing the Middleware to Support UbiComp Applications

Before we describe the architecture and implementation of *GAS-OS*, the proposed middleware for building UbiComp applications, we first want to motivate our design rationale with respect to the challenges and requirements stated above.

GAS-OS implements the concepts encapsulated in GAS [6], a generic architectural style, which can be used to describe everyday environments populated with computational objects. The key idea behind GAS-OS is the uniform abstraction of AmI object services and capabilities via the *plug/synapse* high-level programming model that abstracts the underlying data communications and access components of each part of a distributed system. The basic idea is that users connect at a logical level a service or content provider and a client, and thus compose applications in an ad-hoc, dynamic way. Simply by creating associations between distributed objects, people cause the emergence of new applications, which can enhance activities of work, re-creation or self-expression, rendering their involvement in a natural and abstract way. Furthermore the plug/synapse model serves as a common interfacing mechanism among AmI objects providing the means to create large scale systems based on simple building blocks. Plugs are software classes that make visible the object's properties, capabilities and services to people and to other objects, while synapses are associations between two compatible plugs, which make use of value mappings and are implemented using a message-oriented set of protocols.

Typical middleware platforms address the problem of communication using the Remote Procedure Call (RPC) model. This is not applicable in our case, because each object is autonomous, having no dependencies from fixed centralized nodes. Inspired by Message-Oriented Middleware (MOM) design a fundamental characteristic of GAS-OS is to enable non-blocking message passing. Messaging and queuing allow nodes to communicate across a network without being linked by a private, dedicated, and logical connection. Every node communicates by putting messages on queues and by taking messages from queues. To cope with the requirement to adapt to a broad range of devices even the more resource constraint ones, ideas from micro-kernel design were taken under consideration, where only minimal functionality is located in the kernel, while extra services can be added as plug-ins.

Furthermore, we decided to adopt Java using a JVM layer to assume the responsibility of decoupling GAS-OS from typical local operations like memory management, communication, etc, also providing the requested platform independence. The JVM layer allows the deployment on a wide range of devices from mobile phones and PDAs to specialized Java processors. The proliferation of Java-enabled end-systems makes Java a suitable underlying layer providing a uniform abstraction for the middleware masking the heterogeneity of the underlying AmI objects, networks etc.

Using p2p communication also provides the requested support for dynamic applications over ad-hoc networks. A p2p communication module inside GAS-OS translates the high-level requests/replies into messages and by using low-level p2p networking protocols dispatches them to the corresponding remote service or device capability.

In order to support the definition and realization of collective functionality as well as to ensure the interoperability among the objects, all AmI objects should use a commonly understood language and vocabulary of services and capabilities, in order to mask heterogeneity in context understanding and real-world models. This is tackled by using the GAS Ontology [2] that describes the semantics of the basic terms of our model for UbiComp applications and their interrelations. The term "service" is a fundamental one in this ontology, which contains a service classification, since the AmI objects offer various services and the demand for a semantic service discovery mechanism is evident. Due to facts that artefacts acquire different knowledge and may have limited capabilities, we decided to divide the GAS Ontology into two layers: the GAS Core Ontology (GAS-CO) that represents the necessary common language and the GAS Higher Ontology (GAS-HO) that describes an artefact's acquired knowledge.

3.1 GAS-OS Architecture

The outline of the GAS-OS architecture is shown in Fig.1. Synapses are established at the application layer of the GAS-OS architecture (Fig. 1) using APIs and protocols provided by GAS-OS kernel. The GAS-OS kernel implements the plug/synapse model manifesting the services and capabilities of AmI objects through plugs, while providing the mechanisms (API and protocols) to perform synapses with other AmI objects via the application layer. Synapses can be considered as virtual communication channels that feed the lower communication levels with high-level data. Interfacing with networking mechanisms (transport layer) is done via the Java platform. Data are transmitted through the physical layer to the other end of the synapse where the reverse transformation process is followed. Data departing from a plug are the result of internal processing of an AmI object usually involving sensor data. Data arriving to plugs are usually translated to AmI object behavior (e.g. activate a specific actuator in order to achieve a goal). Using ontologies and the ontology manager plug-in (presented in section 3.2), this translation is done based on the commonly accepted terms of GAS, as encoded in GAS-CO. The resource manager plug-in on the other hand keeps track of available local resources and arbitrates among conflicting requests for those resources. Resources include OS-level resources (memory, CPU, power, etc) as well as high-level resources (sound, light, etc). Through the well-defined interfaces of the plug-in manager, other plug-ins (e.g. security), not currently supported, can be attached to the GAS-OS architecture.

The GAS-OS kernel is the minimum set of modules and functionalities every distributed object must have in order to participate in ubiquitous applications. The GAS-OS Kernel encompasses a Communication Module, a Process Manager, a State Variable Manager, and a Memory Manager as shown in Fig. 1.

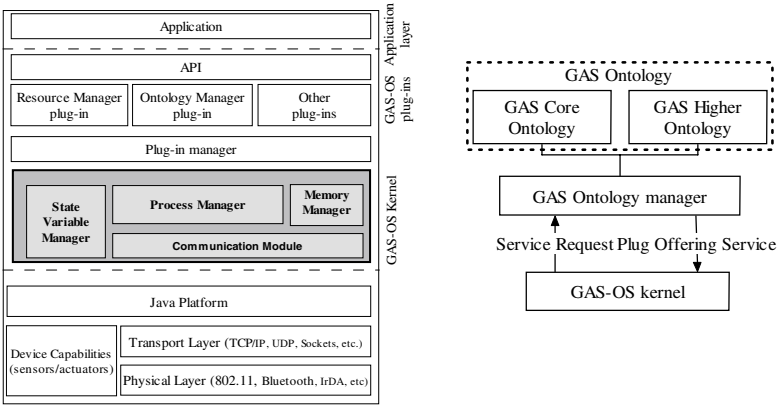

Fig. 1. Left: GAS-OS layered architecture diagram. Right: The GAS Ontology manager

The Communication Module is responsible for communication between GAS-OS nodes. P2P communication is implemented adopting the basic principles and definitions of the JXTA project. Peers, pipes and endpoints are combined into a layered architecture that provides different levels of abstraction throughout the communication process. Peers implement protocols for resource and service discovery, advertisement, routing as well as the queuing mechanisms to support asynchronous message exchange. In order to avoid large messages and as a consequence traffic congestion in the network, XML-based messages are used to wrap the information required for each protocol. Pipes correspond to the session and presentation layers of the ISO-OSI reference model, implementing protocols for connection establishment between two peers, supporting multicast communication for service and resource discovery, while at the same time guaranteeing for reliable delivery of messages. In cases where reliable network protocols are used in the transport layer (e.g. TCP/IP), pipes are reduced to acknowledging for application-level resource availability. Endpoints are associated to specific network resources (e.g. a TCP port). According to the transport layer chosen we can have many different endpoints (e.g. IP-based, Bluetooth, IrDA, etc.), which can also serve as a bridge for different networks. Finally, in order to discover and use services and resources beyond the reachability of wireless protocols (e.g. RF), we have adopted the Zone Routing (ZRP) hybrid routing protocol.

The Process Manager is the coordinator module of GAS-OS. Its most important tasks are to manage the processing policies of GAS-OS, to accept and serve various tasks set by the other modules of the kernel and to implement the Plug/Synapse model. Plugs wrap the information required to describe a service. If the ontology manager plug-in is used a higher-level / contextual description of the service may also be available. Synapses are software entities attached to plugs, wrapping the required information for the remote plug; also having properties that define the interaction patterns (e.g. interaction rules). The process manager implements the protocols required supporting creating, destroying and altering synapses and as a consequence configuring and reconfiguring a UbiComp application consisted of several AmI objects.

The State Variable Manager is a repository of the object's capabilities (e.g. sensors/actuators) inside GAS-OS reflecting at each moment the state of the hardware. An event-based mechanism is used to facilitate communication between the state variable manager and the process manager in order to set up a real time reactive system.

Finally, the Memory Manager enhances the memory management performed by the JVM towards the specific needs of GAS-OS by queuing tasks and messages, buffering sensor and actuator data, storing the state of the AmI object and caching information of other AmI objects to improve communication performance.

3.2 GAS Ontology Manager

The GAS Ontology Manager is the module that manages the GAS Ontology stored at each artefact and implements the interaction of an artefact with its stored ontology. Furthermore, it is responsible to provide to the other modules of GAS-OS any knowledge needed from the ontology. The right part of Fig. 1 demonstrates the interaction among the ontology manager and the GAS-OS kernel. An important feature of the ontology manager is that it adds a level of abstraction between GAS-OS and the GAS ontology, meaning that only the ontology manager can understand and manipulate the ontology; the GAS-OS can simply query this module for information stored into the ontology without having any knowledge about the ontology language or its structure.

Since GAS-CO must be common for all artefacts and cannot be changed during the use of UbiComp applications, this module provides only methods for acquisition of knowledge, such as the definitions of basic concepts and the service classification. Likewise it can only query the GAS-HO-static of an artifact. On the other hand, as it is responsible for keeping up to date an artefact's GAS-HO-volatile, it can both read and write it. As the GAS-HO contains only instances of the concepts defined in the GAS-CO, the basic methods of the ontology manager relevant to the GAS-HO can query for an instance and add new ones based on the concepts defined in the GAS-CO. Thus an important feature of the GAS Ontology manager is that it enforces the integrity of the instances stored in the GAS-HO with respect to the concepts described in GAS-CO.

The interoperability among AmI objects is initially established using the objects' GAS-HO; if their differences lead to infeasible interoperability, each local GAS Ontology manager is responsible for the interpretation of different GAS-HOs based on the common GAS-CO. Thus the semantic interoperability among AmI objects is greatly improved. The GAS Ontology manager also enables knowledge exchange among AmI objects by sending parts of an object's GAS-HO to another object.

One of the ontology's goals is to describe the services that artefacts provide so that to support a service discovery mechanism. Thus the ontology manager provides methods that query the ontology for the services that an artefact offers as well as for artefacts that provide specific services. The GAS-OS get from the ontology manager the necessary knowledge stored in an AmI object's ontology relevant to its services, in order to implement a service discovery mechanism. Finally the GAS Ontology manager using this mechanism and the service classification can identify

AmI objects that offer similar semantically services and propose objects that can replace damaged ones. So it supports the deployment of adaptive and fault-tolerant UbiComp applications.

4 Building a Real Life Home Application

This section demonstrates the development of a real life application starting from a high level description of the target scenario to its implementation based on the services offered by the GAS-OS middleware.

Let's take a look at the life of Patricia, a 27-year old woman, who lives in a small apartment near the city centre and studies Spanish literature. A few days ago she heard about these new AmI objects and decided to give herself a very unusual present: a few furniture pieces and other devices that would turn her apartment into a smart one! On the next day, she was waiting for the delivery of an eDesk (sensing light intensity, temperature, weight on it, proximity of a chair), an eChair (could tell whether someone was sitting on it), a couple of eLamps (could be remotely turned on and off), and some eBook tags (could be attached to a book, telling whether it is open or closed and determine the amount of light that falls on it). Pat had asked the store employee to pre-configure some of the artifacts, so that she could create a smart studying corner in her living room. Her idea was simple: when she sat on the chair and she would draw it near the desk and then open a book on it, then the study lamp would be switched on automatically. If she would close the book or stand up, then the light would go off.

The behavior requested by Pat requires the combined operation of the following set of AmI objects using their plugs: eDesk (Reading, Proximity), eChair (Occupancy), eLamp (Light_Switch) and eBook (Open/Close). Then a set of synapses has to be formed, for example, associating the Occupancy plug of the eChair and the Open/Close plug of the eBook to the Proximity plug of the eDesk, the Reading plug of the eDesk to the Light_Switch plug of the e-Lamp, etc.

The capability of making synapses is a service offered by GAS-OS and is implemented in simple steps as described below. Consider the synapsing process among the Reading plug of the eDesk and the Light_Switch plug of the eLamp.

Initially, the eDesk sends a *Synapse Request* message to the eLamp containing information about the eDesk and its Reading plug as well as the id of the Light_Switch plug. Then the eLamp activates the *Synapse Response* process by first checking the plug compatibility of the Reading and Light_Switch plugs, to confirm that they are not both service providers only (output plugs) or both service receptors only (input plugs). If the compatibility test is passed, an instance of the Reading plug is created in the eLamp (as a local reference) and a positive response is sent back to the eDesk. The instance of the Reading plug is notified for changes by its remote counterpart plug and this interaction serves as an intermediary communication channel. In case of a negative plug compatibility test, a negative response message is sent to the eDesk. Upon a positive response, the eDesk also creates an instance of the Light_Switch plug, and the connection is established (Fig. 2-left). After connection's establishment, the two plugs are able of exchanging data, using the *Synapse Activation* mechanism.

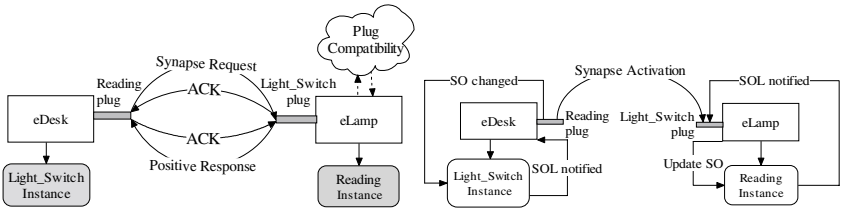

Fig. 2. Synapsing process between Reading and Light_Switch plugs

Output plugs (Reading) use shared objects (SO) to encapsulate the plug data to send, while input plugs (Light_Switch) use event-based mechanisms, called shared object listeners (SOL), to become aware of incoming plug data. When the value of the shared object of the Reading plug changes the instance of the Light_Switch plug in the eDesk is notified and a synapse activation message is sent to the eLamp. The eLamp receives the message and changes the shared object of its Reading plug instance. This, in turn, notifies the target Light_Switch plug, which reacts as specified (Fig. 2-right). Finally, if one of the two connected plugs breaks the synapse, a *Synapse Disconnection* message is sent to the remote plug in order to also terminate the other end of the synapse.

But how are the above messages actually exchanged between AmI objects? In the example, both the eDesk and the eLamp own a communication module with an IP-based (dynamically determined) Endpoint. Plug/Synapse interactions (e.g. synapse establishment) are translated to XML messages by the communication module and delivered to the remote peer at the specified IP address (Fig. 3).

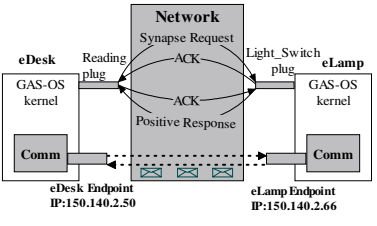

Fig. 3. From Plug/Synapse interactions to p2p communication

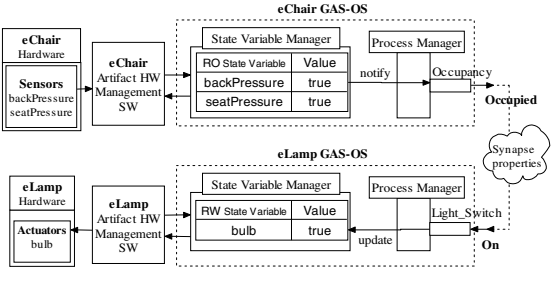

Fig. 4. Communication with hardware

Having described the ways interaction among objects is implemented using GAS-OS, what is missing to close the loop is interaction with the end-users of the Ubi-Comp application. This interaction is done via the sensors and actuators each artefact has, but the way the sensing and actuating data are manipulated by each object is also facilitated by GAS-OS and specifically through its State Variable Manager. In Fig. 4, the eChair has two pressures sensors (back, seat) to sense that someone is sitting on it, and the eLamp has one bulb actuator, both reflected inside GAS-OS as state variables in the SVM. Through communication with the eChair hardware management software the eChair's SVM retrieves all the sensor information of the eChair and registers itself as a listener for changes of the environment. It also communicates with the Process Manager to promote the eChair-eLamp communication as it feeds the Weight plug with new data coming from the hardware that results in the Weight-Light synapse. The reverse process is followed on the other end of the synapse. The matching of the "Occupied" / "Not Occupied" values of the Occupancy plug with the "On" / "Off" states of the Light_Switch plug, is done by configuring the properties of the synapse. So by mapping the "Occupied" state of the eChair to the "On" state of the eLamp and the "Not_occupied" to "Off" we have the following (desired) behavior: "sitting on the chair switches the lamp on while leaving the chair switches the lamp off".

5 System Evaluation

The primary goal for GAS-OS was to serve as a proof of concept that users could be enabled to configure UbiComp applications by using everyday objects as components. Following we discuss if GAS-OS meets the demands of UbiComp applications.

GAS-OS proved capable of running on different devices, satisfying our requirement for supporting resource constrained devices. Devices were developed ranging from handheld computers running Win CE and Java PE, to COTS java-based boards like the EJC [10]. GAS-OS was also tested on devices using Microsoft UPNP protocol to communicate with their hardware, running on the SNAP embedded J2ME controller [12], by interfacing UPNP with GAS-OS. The overall integration process proved easy enough, providing strong indications concerning the independence of GAS-OS from communication protocols and its interfacing ability with standards.

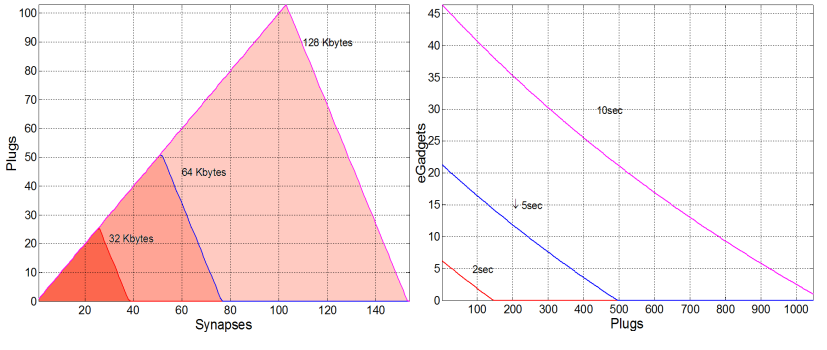

Fig. 5. L.: Max synapses when constraining memory vs # plugs that can participate. R.: # objects that can be discovered in a certain period of time vs # of plugs.

Measuring the process of discovering artefacts provided feedback on the efficiency of GAS-OS to discover a certain number of objects within a time frame and consequently an estimation of how long will the user have to wait in order to discover his ubiquitous environment. (Fig.5-right) shows the number of objects that can be discovered in successive time intervals, versus the number of plugs. To reach maximum performance overhead, we have to get to a large number of plugs per object.

As plugs and synapses are the main factors that increase memory requirements during the execution of an application, we studied the relation between the number of plugs and synapses that participate for constraint amounts of memory. Maximum memory allocation is achieved when each plug participates in only one synapse (Fig.5-left). The more plugs participating in a synapse, the more the allocated memory until we reach the memory constraint. From this point and on more synapses can only be achieved if distributed to fewer plugs.

Using code instrumentation, we measured the average synapsing and communication time in an application where 5 objects are inter-connected with 6 synapses. After creating the 1st synapse only a few milliseconds are required to create the rest, while the average time of approximately 1 sec for all 6 synapses is acceptable. For communication among 2 objects having a synapse, the average time is only a few milliseconds, which is acceptable for real time applications. These measurements include the overhead of the 802.11b protocol, while messages exchanged vary from a few bytes to 1 KB. What is important though is that after synapses' establishment communication between objects is fast, satisfying our requirement for real time response.

The use of ontologies in order to deal with heterogeneity in service definition improved interoperability of objects. Specifically the service discovery mechanism enabled the identification of semantically similar services and GAS-OS, exploiting this, could replace a failed or moved AmI object with a similar one in satisfactory time.

Finally, GAS-OS, the end-user programming tools (GAS Editor) that use it and the applications built with it, were evaluated in user and expert trials [7]. During the development and deployment of UbiComp applications from both novice and experienced users, we got fairly encouraging results regarding usability, as using GAS Editor it was proven easy to create, configure and reconfigure UbiComp applications.

6 Related Work

Several research efforts are attempting to design ubiquitous computing architectures. In "Smart-Its" [4] the aim is to develop small devices, which, when attached to objects, enable their association based on the concept of "context proximity". The collective functionality of such system is composed of the computational abilities of the Smart-Its, without taking into account the "nature" of the participating objects. A more generic approach is the one of "Oxygen" [11], which enables human-centered computing by providing special computational devices, handheld devices, dynamic networks, etc. The "Accord" [9] focuses on developing a Tangible Toolbox (based on the idea of tangible puzzle) that enables people to easily embed functionality into existing artefacts around home and permit artefacts to be integrated with each other.

The GAIA system [8] provides an infrastructure to spontaneously connect devices offering or using registered services. GAIA-OS requires a specific system software

infrastructure using CORBA objects, while mobile devices cannot operate autonomously without this infrastructure. In GAIA is used an ontology server that maintains various ontologies, addressing issues like service discovery and context-awareness; a fairly different approach from ours. The BASE [1] is a component-oriented microkernel based middleware, which, although provides support for heterogeneity and a uniform abstraction of services, the application programming interface requires specific programming capabilities by users. Finally TinyOS [3] is an event driven operating system designed to provide support for deeply embedded systems, which requires concurrency intensive operations while constrained by minimal hardware resources.

7 Conclusions

In this paper we presented GAS-OS, a middleware for building UbiComp applications from individual artifacts using the plug/synapse abstraction layer. GAS-OS, being a component framework, determines the component interfaces and the rules governing their composition, and provides a clear separation between computational and compositional aspects of such applications, leaving the latter to ordinary people, while the former can be undertaken by designers or engineers. End-users only have to compose their applications as instances of the system. As a component-based application can be reconfigured to meet new requirements at a low cost, composition achieves adaptability and evolution. The possibility to reuse objects for purposes not accounted for during the design opens roads for emergent uses of artefacts that result from actual use.

References

1. Becker C. et al., “BASE - A Micro-broker-based Middleware For Pervasive Computing”, in Proceedings of the 1st IEEE International Conference on Pervasive Computing and Com-munication (PerCom03), Fort Worth, USA, 2003.
2. Christopoulou E., Kameas A., “GAS Ontology: an ontology for collaboration among ubiquitous computing devices”, International Journal of Human – Computer Studies, Vol. 62, issue 5, Protégé: Community is Everything (2005), pp 664-685, Elsevier Ltd.
3. Hill J. et al., “System architecture directions for networked sensors”, In Architectural Support for Programming Languages and Operating Systems. (2000) 93-104
4. Holmquist L.E. et al., “Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artifacts”, in Proc. of UbiComp 2001, Atlanta, USA, Sept. 2001.
5. IST Advisory Group, “Scenarios for Ambient Intelligence in 2010-full”, February 2001.
6. Kameas A. et al., “An Architecture that Treats Everyday Objects as Communicating Tangible Components”, in Proc. of the 1st IEEE PerCom, Fort Worth, USA, 2003.
7. Markopoulos P., Mavrommati I., Kameas A., “End-User Configuration of Ambient Intelligence Environments: Feasibility from a User Perspective”, In the proc. of the 2nd European Symposium on Ambient Intelligence, LNCS 3295, pp. 243-254, November 2004.
8. Román M., Campbell R.H., “GAIA: Enabling Active Spaces”, Proceedings of the 9th ACM SIGOPS European Workshop, pp. 229-234, Kolding, Denmark, September 2000
9. Accord project website: http://www.sics.se/accord/home.html
10. EJC website: http://www.embedded-web.com/
11. Oxygen project website: http://oxygen.lcs.mit.edu/
12. Simple Network Application Platform (SNAP) website: http://snap.imsys.se/