A CONCEPTUAL MODEL AND THE SUPPORTING MIDDLEWARE FOR COMPOSING UBIQUITOUS COMPUTING APPLICATIONS

N. Drossos[1], C. Goumopoulos[1] and A. Kameas[1, 2]

[1]Computer Technology Institute, Hellas [2]Hellenic Open University, Hellas

ABSTRACT

Given the resulting complexity of the ambient applications that one can form in the Ubiquitous or Pervasive Computing domain it is required to abstract the intrinsic characteristics of specific communication models away from the application logic. These applications will be characterized by the increasing ubiquity of interactions between many possibly heterogeneous artifacts and services. This paper presents the Plug/Synapse abstraction, which provides a conceptual model for building ubiquitous computing applications in a high-level programming manner. GAS-OS is the software layer that implements the Plug/Synapse model and the concepts encapsulated in GAS, a generic architectural style, which can be used to describe everyday environments populated with computational artifacts. The paper focuses on the design and architecture of GAS-OS, which is the minimum set of modules and functionalities that every device must have, in order to be a ubiquitous computing artifact and participate in artifact collections.

1. INTRODUCTION

The vision of Ambient Intelligence (AmI) implies a seamless environment of computing, advanced networking technology and specific interfaces ISTAG 1). 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 pro-vided by the environment and the services provided by the objects therein.
An important characteristic of AmI environments is the merging of physical and digital space (i.e. tangible objects and physical environments are acquiring a digital representation). As the computer disappears in the environments surrounding our activities, the objects therein become augmented with Information and Communication Technology (ICT) components (i.e. sensors, actuators, processor, memory, wire-less communication modules) and can receive, store, process and transmit information; in the following, we shall use the term “artifacts” for this type of augmented objects. These objects may be new or improved versions of existing objects, which by using the new technology, allow people to carry out new or old tasks in new and better ways. The provision of conceptual models and software tools for creating, managing, communicating with, and reasoning about, these new ecologies (or UbiComp applications) is of paramount importance, because people involvement is considered crucial for the successful adoption of this new computing paradigm.
This paper introduces the Plug/Synapse model and the GAS-OS middleware. The former provides a conceptual model for building ubiquitous computing applications in a high-level programming manner. These are considered to consist of tangible objects (artifacts), which carry the computing and networking technology required. The latter is embedded in these artifacts and supports their collective operation. This model is part of the Gadgetware Architectural Style (GAS), which constitutes a generic framework, shared by users and designers, for consistently describing, using, reasoning about UbiComp applications within the AmI environment.
The rest of the paper is organized as follows. Section 2 goes through the Plug/Synapse model and offers an example of everyday life scenario, before the design and architecture of the software that implements and validates the model described in section 3. Section 4 presents some implementation details based on the mentioned scenario, while section 5 provides a performance evaluation of the software before existing approaches and conclusions are discussed in section 6.

2. THE ABSTRACTION LAYER: THE PLUG/SYNAPSE MODEL FOR COMPOSING UBICOMP APPLICATIONS

Within an AmI environment, a UbiComp application may be composed of a number of heterogeneous artifacts or devices, which may be stationary or portable. Those artifacts and devices have different, dynamically changing capabilities and specific ways to use them; yet, all of them can communicate.
By providing uniform abstractions and a supporting middleware, we treat objects as components of a UbiComp application. In this approach, a digital counterpart of the properties of artifacts and devices is created, while making them composeable, thus enabling their association in order to achieve synthetic functionality. The Plug/Synapse model, which can be seen as an extension to the client-server model, is based on the jigsaw metaphor and has appeared to end-users to

be intuitive to use. 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 everyday artifacts, 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.

The basic concepts encapsulated in our model are:

- ***Plugs***: From a user's perspective, they make visible the artifacts' properties, capabilities and services to people and to other artifacts; they are implemented as software classes
- ***Synapses***: They are associations between two compatible plugs, which make use of value mappings; they are implemented using a message-oriented set of protocols

We assume that no specific networking infrastructure exists, thus ad-hoc networks are formed. The network interfaces used are highly heterogeneous ranging from infra-red communication over radio links to wired connections. Since every node serves both as a client and as a server (devices can either provide or request services at the same time), communication between artifacts can be considered as peer-to-peer Schollmeier 2.

Given the resulting complexity of the UbiComp applications that one can form in an AmI environment, it is required to abstract the intrinsic characteristics of specific communication models away from the application logic. The Plug/Synapse model is independent of the underlying protocols, needed for example to route messages or to discover resources in realization of an application. It only requires that artifacts are able to communicate and they have to run GAS-OS in order to "comprehend" each-other, so that people can access their services, properties and capabilities in a uniform way. People in that way would not need to be engaged in any type of formal "programming" in order to achieve the desired functions. The application is realized through the cooperation of artifacts in the form of established synapses between plugs. The approach adopted is that people live in an environment populated with artifacts; they have a certain need or task, which they think can be met or carried out by (using) a combination of services and capabilities; then, they search for artifacts offering these services and capabilities as plugs; they select the most appropriate ones and combine the respective plugs into functioning synapses; if necessary, they manually adapt or optimize the collective functionality.

For example, let's take a look at the life of Patricia, a 27-year old single woman, who lives in a small apartment near the city centre and studies Spanish literature at the Open University. A few days ago she passed by this store, where she saw an advertisement about these new augmented artifacts, the "extrovert Gadgets". Pat decided to enter. Half an hour later she had given 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 anxiously waiting for the delivery of an eDesk (it could sense light intensity, temperature, weight on it, proximity of a chair), an eChair (it could tell whether someone was sitting on it), a couple of eLamps (one could remotely turn them on and off), and some eBook tags (they could be attached to a book, tell whether a book is open or closed and determine the amount of light that falls on the book). 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 artifacts: eDesk, eChair, eLamp and eBook. Some indicative plugs for each of these artifacts are: 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 e-Desk to the Light_Switch plug of the e-Lamp, etc.

3. THE SYSTEM LAYER: GAS-OS MIDDLEWARE

The key idea behind GAS-OS is the uniform abstraction of artifact 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. Inspired by 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 need to adapt to a broad range of devices, we adapted ideas from micro-kernel design Tanenbaum et al 3) where only minimal functionality is located in the kernel, while extra services can be added as plug-ins.

In order to maintain the autonomous nature of artifacts and at the same time make even the more resource constraint ones capable of participating in ubiquitous applications, the kernel of GAS-OS is designed to support only accepting and dispatching messages, managing local hardware resources (sensors/actuators), and the plug/synapse interoperability protocols. Fig. 1 shows GAS-OS being embedded into the high-level design of an artifact. A Sensor / Actuator network together with custom FPGA or PIC micro-controller based boards are responsible for converting artifact data (e.g. pressure, luminosity etc.) to digital ones and vice versa Kameas et al 4. The Java Virtual Machine (JVM) layer assumes the responsibility of decoupling GAS-OS from typical local operations like memory management, serial and RF communication, etc.

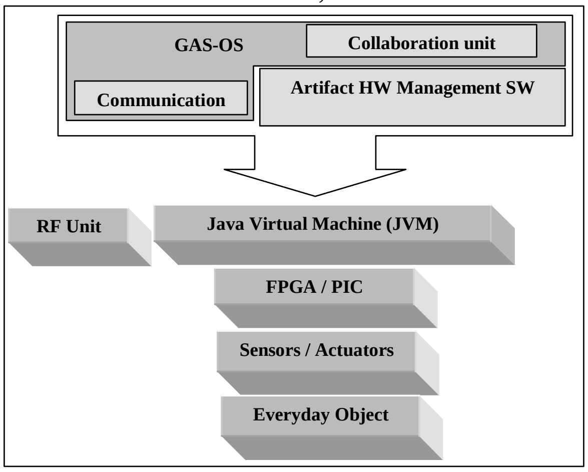


Fig.1. High level design of an artifact

Consequently, the JVM bridges the hardware – software gap, by passing sensor data from the hardware level, to the upper software entities. Digital data must be analyzed, grouped and treated in a different way for each artifact; thus, a specific artifact hardware management module, playing the role of a software driver, must be implemented per artifact.

The GAS-OS kernel implements plugs and by using the plug/synapse interoperability protocols it can initialize or participate in a synapsing process. The communication module translates the high-level requests/replies into messages and by using low-level peer-to-peer networking protocols, it dispatches them to the corresponding remote service or device capability. The kernel is also capable of managing service and resource discovery messages in order to facilitate the formation of the proper synapses.

In order to support the definition and realization of collective functionality, all artifacts should use a commonly understood vocabulary of services and capabilities, in order to mask heterogeneity in context understanding and real-world models. This has been implemented as a multi-level ontology, coupled with a GAS-OS mechanism that will be able to handle the knowledge described by ontology and use it in order to facilitate the communication between two or more artifacts Christopoulou and Kameas 5.

Finally, the Java Virtual Machine 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. The proposed high level design hides the heterogeneity of the underlying artifacts, sensors, networks etc. (masked by JVM and GAS-OS) and provides the means to create large scale systems based on simple building blocks.

GAS-OS Architecture

The outline of the GAS-OS architecture is shown in Fig.2. The GAS-OS kernel implements the plug/synapse model manifesting the artifact's services and capabilities through plugs, while providing the mechanisms (API and protocols) to perform synapses with other artifacts via the application layer. Synapses can be considered as virtual channels that feed the lower communication levels with high-level data. Interfacing with networking mechanisms (transport layer) is done via the Java platform and finally data are transmitted through the physical layer to the other end of the synapse to follow the reverse process of transforming low-level information (e.g. messages) to high-level one (e.g. service requests). Data departing or arriving to plugs usually affect one or more of the device capabilities (sensors/actuators), while the kernel assumes the responsibility of translating those data to artifact behavior (e.g. activate a specific actuator in order to achieve a goal).

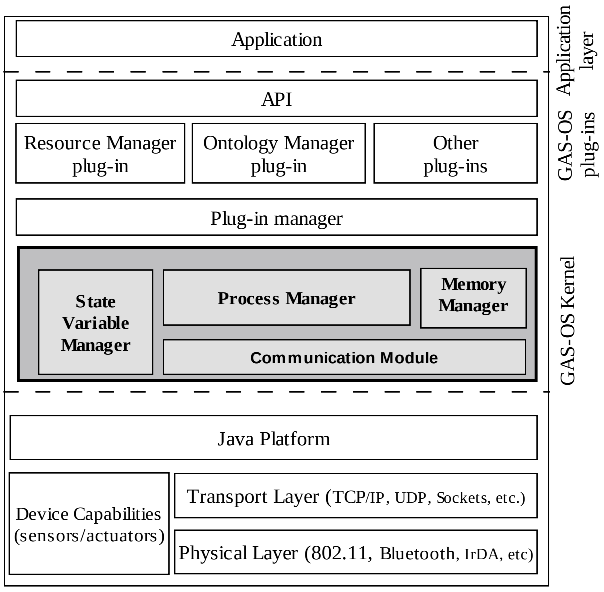


Fig. 2. GAS-OS layered architecture diagram

The GAS-OS kernel encompasses a Communication Module, a Process Manager, a State Variable Manager, and a Memory Manager as shown in Fig 2. The Communication Module is responsible for communication between different GAS-OS nodes. This module implements algorithms and protocols for wireless, connectionless communication as well as mechanisms for internal diffusion of information exchanged. The Process Manager is the coordinator module of GAS-OS. Some of its most important tasks are to manage the processing policies of the GAS-OS, to accept and serve various tasks set by the other modules of the kernel and to implement the Plug/Synapse model. The State Variable Manager is a repository of the

hardware environment (sensors/actuators) inside GAS-OS reflecting at each particular moment the state of the hardware. The Memory Manager is responsible for handling the memory resources of an artifact, storing its state and caching information of other artifacts to improve communication performance.
Using ontologies and the ontology manager plug-in, artifacts can obtain context-awareness and manifest higher-level behavior. Applications state their resource or service needs through concepts that are part of the artifact's ontology. In that way, high-level descriptions of services and resources independent of the context of a specific application are possible Holmquist et al 6, facilitating the mutual understanding between heterogeneous artifacts as well as the discovery of services. The resource manager plug-in on the other hand, encounters physical resources scarcities by providing resource mapping and mediation services. The basic function it performs is to keep track of available resources and arbitrate among conflicting requests for those resources. Re-sources include OS-level resources (memory, CPU, power, etc) as well as high-level resources such as sound, light, etc. Finally through the well-defined interfaces of the plug-in manager, other plug-ins (e.g. security) can be easily attached to the GAS-OS architecture.

4. IMPLEMENTATION

The current version of GAS-OS has been implemented in the Java Personal Edition (PE) that is fully compatible with the Java Standard Edition 1.1.8. Although Sun has abandoned the further development of Java PE, J2ME Personal Profile allows all Java PE programs to execute in J2ME compliant devices. So far, GAS-OS has been tested in laptops, IPAQs and finally in the EJC (Embedded Java Controller) board EJC 7. This decision provides platform independence and allows us to run GAS-OS on a multitude of different devices.
The following sections will describe implementation details concerning three basic functions supported by GAS-OS in order to realize ubiquitous computing applications using the example introduced in section 2. Synapse management handles the process of associating logical channels (synapses) among artifacts, while inter-artifact communication supports the formation and operation of synapses at the network layer by establishing peer-to-peer connections over various physical layers. Finally, interfacing with the artifact's hardware describes how GAS-OS handles sensors and actuators of an artifact in order to satisfy the high-level behavior dictated by the association with other artifacts using the plug/synapse model.

4.1. Synapse Management

The management of synapses is performed by the Process Manager module. The Process Manager collaborates with the Communication Module and the State Variable Manager (Fig. 6), and sets up an event based internal messaging system that combine input from sensors and actuators with input received wirelessly from other artifacts. Furthermore, it implements the plug/synapse model and specifies the process for building a synapse.
Consider the synapsing process among the Reading plug of the eDesk and the Light_Switch plug of the eLamp:

- *Synapse Request*: The eDesk sends a "Connection Request" message to the eLamp. The message contains information concerning the eDesk and its Reading plug as well as the id/name of the Light_Switch plug.

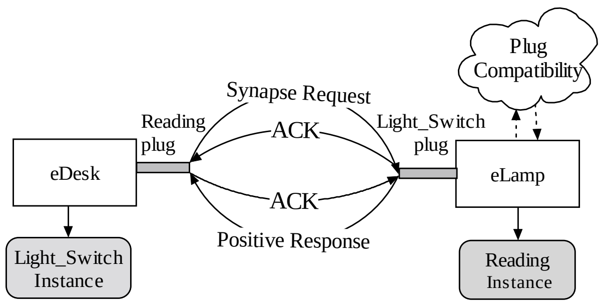


Fig. 3. Synapse establishment between plugs Reading and Light_Switch

- *Synapse Response*: When the eLamp receives the message it first checks the plug compatibility of the Reading and Light_Switch plugs. Plug compatibility lies in confirming that the two plugs are not both service providers only (output plugs) or both service receptors only (input plugs). In the example the Reading plug is output and the Light_Switch plug is input, so the compatibility test is passed, and 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, while no instance of the Reading plug is created. When the eDesk receives a positive response, it also creates an instance of the Light_Switch plug, and the connection is established. Fig. 3 summarizes the whole procedure.

- *Synapse Activation*: After connection established, the two plugs are capable of exchanging data. Output plugs (Reading) use specific objects, called shared objects (SO), to encapsulate the plug data to send, while input plugs (Light_Switch) use specific 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. 4).

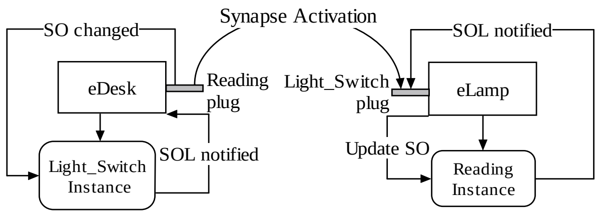


Fig. 4. Synapse activation

- *Synapse Disconnection*: 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.

4.2. Inter - Artifact Communication

The Communication Module is responsible for communication between different artifacts. This module, implements protocols for wireless, connectionless communication as well as mechanisms for internal diffusion of information exchanged. Peer-to-peer 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 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 (e.g. sending synapse request message to an incompatible plug will return a NACK message). Endpoints are considered as the fundamental networking units and 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 Pearlman and Haas 8. ZRP is a hybrid approach combining a proactive and a reactive part, trying to minimize the sum of their respective overheads and scales very well when the traffic or the mobility is increased.

In the example introduced in section 2, 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.

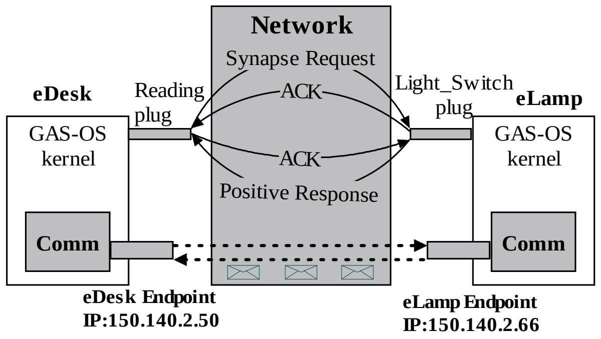


Fig. 5. From Plug/Synapse interactions to p2p communication

4.3. Interfacing with the Artifact Hardware

The interfacing of the artifact with its hardware (sensors/actuators) is performed as collaboration between the artifact hardware management software and the State Variable Manager module of the GAS-OS. The State Variable Manager (SVM) holds two separate structures, one for the Read Only (RO) and one for the Read Write (RW) state variables. State Variables reflect the state of an artifact's hardware, like sensors and actuators. For example 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 (Fig. 6).
Through communication with the eChair hardware management software (Fig. 6) the eChair's SVM retrieves all the sensor information of the eChair and registers itself as a listener for changes of the environment. Moreover, it communicates with the Process Manager to promote the eChair-eLamp communication as it feeds the Weight plug with new data coming from the hardware, which finally result in the Weight-Light synapse. On the other end of the synapse the eLamp receives data from the Light_Switch plug and through the Process Manager the data are translated to low level actuator data, resulting in the eLamp's bulb actuator.

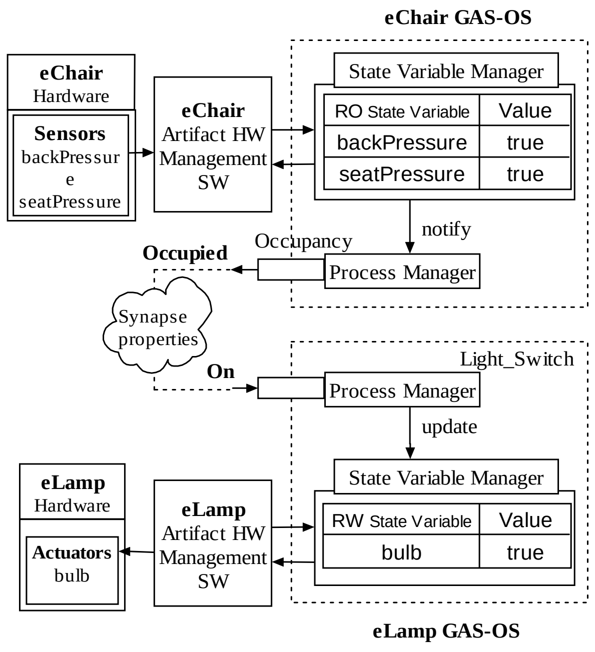


Fig. 6. Communication with hardware

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 for example 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”. The Mappings structure holds records where the key is the Synapse itself and the content is a number of values-to-states mappings. The Process Manager uses these mappings to filter the incoming information from input plugs and give a specific meaning to the incoming data.

5. PERFORMANCE EVALUATION

To estimate the performance of GAS-OS and its appropriateness to support the execution of ubiquitous applications, a performance and scalability analysis based on theoretical analysis as well as on experimental data was carried out. The results involve the memory requirements of GAS-OS and the throughput for the case of eGt discovery in relation to the available services (plugs). Finally a pure experimental measurement of the synapsing process and communication takes place in order to obtain an indication of the time required to set up a ubiquitous application.
The code size of the current implementation of GAS-OS is approximately 200 KB. Measuring the memory footprint is crucial in order to indicate that GAS-OS can be executed on resource constraint devices. We measured the memory footprint of the GAS-OS kernel running upon the Sun Personal Java on a Compaq IPAQ PDA reference system. First, measurements were done by instrumented special measurement code inside GAS-OS and second using the JProbe Memory Profiler tool JProbe 9. In both cases results seem to converge to approximately 23Kbytes of memory, while during runtime more memory may be allocated depending on the application (e.g. number of plugs, synapses, device capabilities etc.).

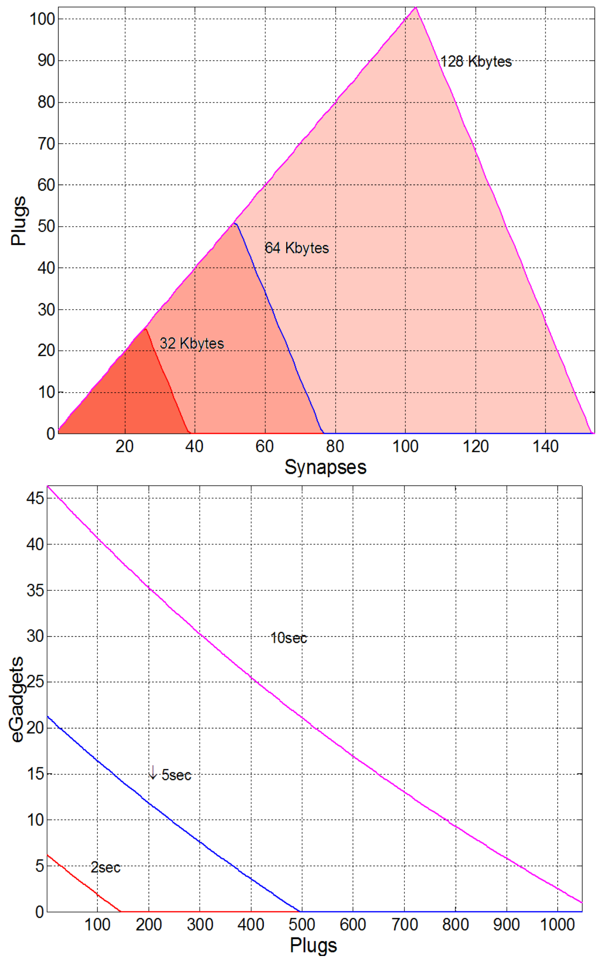
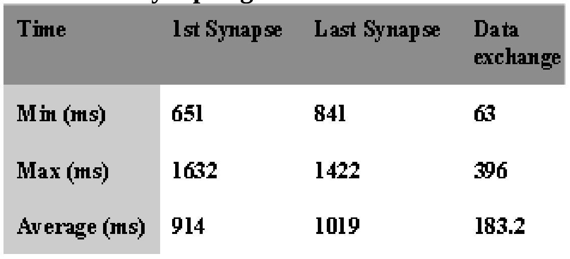


Fig. 7. Up: Maximum number of synapses when constraining memory versus the number of plugs that can participate. Down: Number of eGts that can be discovered in a certain period of time versus the number of plugs.

As plugs and synapses are mainly what increases memory during the execution of an application, we studied the relation between the number of plugs and the number of synapses that participate for constraint amounts of memory. Maximum memory allocation is achieved when each plug participates in only one synapse (Fig. 7 Up). The more plugs participating in one synapse, the more the allocated memory until we reach the memory constraint. From this point and on (peaks) more synapses can only be achieved if distributed to fewer plugs.
In order to measure the throughput of GAS-OS we consider the process of discovering eGts with a certain number of plugs. Studying the discovery process gives an indication of the number of eGts that will be

discovered in a certain period of time, and as a consequence how long will the user have to wait in order to discover his ubiquitous environment. The number of eGts that can be discovered in successive time intervals, versus the number of plugs (Fig. 7 Down): in order to have maximum performance overhead, we have to get to a large number of plugs per eGt.

TABLE 1 - Synapsing and Communication Times



Time	1st Synapse	Last Synapse	Data exchange
Min (ms)	651	841	63
Max (ms)	1632	1422	396
Average (ms)	914	1019	183.2

Min, Max and average times in milliseconds to create the 1st and the last synapse in a GadgetWorld of 5 eGts with a total of six synapses. After creating the 1st synapse only a few milliseconds are required to create the rest of the synapses, while the average time of approximately 1 sec for all six synapses is acceptable. For communication between 2 eGts having a synapse, the average time is only a few milliseconds, which is acceptable for real time applications.

Using code instrumentation, we measured the average time for making a synapse and for communicating in an application where five eGts are inter-connected with six synapses (Table 1). These measurements include the overhead of the IEEE 802.11b protocol, while messages exchanged vary from a few bytes to 1 Kbyte. Synapse times refer to the amount of time needed from the point the user specifies a synapse up to the time this synapse is completed. In cases where the eGts, specified to form a synapse, are not aware of each other, a discovery phase is also included in the overall synapsing process. Thus, the min synapse time refers to a synapse without a discovery, while the max to a synapse with a discovery overhead. It is important that after synapses are established (GW set-up) communication between eGts is fast, satisfying our requirement for real time response.

6. CONCLUSIONS

Several research efforts are attempting to design ubiquitous computing architectures. Project "Smart-Its" 6 aims at developing small devices, which, when attached to objects, enable their association based on the concept of "context proximity". Thus, the collective functionality of such a system is mainly composed of the computational abilities of the Smart-Its, without taking into account the "nature" of the participating objects. A more complete and generic approach is undertaken by project "Oxygen", which enables human-centered computing by providing special computational devices, handheld devices, dynamic networks and other supporting technologies Oxygen 10. Another interesting project is "Accord", which is focused in developing a Tangible Toolbox (based on the metaphor of a tangible puzzle) that will enable people to easily embed functionality into existing artefacts around the home and enable these devices to be integrated with each other Accord 11. Other related research efforts are:

- Gaia Román and Campbell 12 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 the infrastructure;
- BASE Becker et al 13 is a component-oriented micro-kernel based middleware, which, although provides support for heterogeneity and a uniform abstraction of services, the application programming interface requires specific programming capabilities by users;
- TinyOS Hill et al 14, an event driven operating system, designed to provide support for deeply embedded systems (i.e. sensor networks), which require concurrency intensive operations while constrained by minimal hardware resources;

The overall innovation of the Gadgetware Architectural Style (GAS) approach lies in viewing the process where people configure and use complex collections of interacting artifacts, as having much in common with the process where system builders design software systems out of components. In the proposed approach, the Plug/Synapse model provides a high-level abstraction of the component interfaces and the composition procedure.

Then, GAS-OS, the software that implements GAS, can be considered as a component framework that determines the interfaces that components may have and the rules governing their composition. GAS-OS manages resources shared by artifacts, and provides the underlying mechanisms that enable communication (interaction) among artifacts. For example, the proposed concept supports the encapsulation of the internal structure of an artifact and provides the means for composition of an application, without having to access any code that implements the interface. Thus, this approach provides a clear separation between computational and compositional aspects of an application, leaving the second task to ordinary people, while the first can be undertaken by experienced designers or engineers.

The benefit of this approach is that, to a large extent, system design is already done, because the domain and system concepts are specified in the generic architecture; all people have to do is realize specific instances of the system. Composition achieves adaptability and evolution: a component-based application can be reconfigured with low cost to meet new requirements. The possibility to reuse devices for several purposes - not all accounted for during their design - opens possibilities for emergent uses of ubiquitous devices, whereby the emergence results from actual use.

REFERENCES

1. ISTAG in FP6: Working Group 1, IST Research Content, Final Report, http://www.cordis.lu/ist/istag.htm

2. Schollmeier R.,2001,"A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications", Proceedings of the First International Conference on Peer-to-Peer Computing P2P'01

3. Tanenbaum A.S., et al,August 1991,"The Amoeba Distributed Operating System-A Status Report", Computer Communications, vol. 14, no. 6, pp. 324-335

4. Kameas A., et al,2003,"An Architecture that Treats Everyday Objects as Communicating Tangible Components", in Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom03), Fort Worth, USA

5. Christopoulou E., Kameas A., "GAS Ontology: an ontology for collaboration among ubiquitous computing devices", to appear in 2004 Special Issue on Protégé the International Journal of Human – Computer Studies

6. Holmquist L.E., et al,Sept. 2001,"Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artifacts", in Proceedings of UBICOMP 2001, Atlanta, GA, USA

7. EJC website: http://www.embedded-web.com/

8. Pearlman M. and Haas Z.,August 1999, "Determining the Optimal Configuration for the Zone Routing Protocol", IEEE Journal on Selected Areas in Communications, Vol. 17, No 8

9. JProbe website: http://www.quest.com/jprobe/index.asp

10. Oxygen project website: http://oxygen.lcs.mit.edu/

11. Accord project website: http://www.sics.se/accord/home.html

12. Román M. and Campbell R.H., September 2000,"GAIA: Enabling Active Spaces", Proceedings of the 9th ACM SIGOPS European Workshop, pp. 229-234, Kolding, Denmark

13. Becker C., et al,2003,"BASE - A Micro-broker-based Middleware For Pervasive Computing", in Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communication (PerCom03), Fort Worth, USA

14. Hill J, et al,2000,"System architecture directions for networked sensors." in Proceedings of ACM Architectural Support for Programming Languages and Operating Systems conference