An Architecture for Ambient Computing

Christophe Jacquet[1,2] Christophe.Jacquet@supelec.fr
Yolaine Bourda[1] Yolaine.Bourda@supelec.fr
Yacine Bellik[2] Yacine.Bellik@limsi.fr

[1]Supélec
Plateau de Moulon
F-91192 Gif-sur-Yvette Cedex

[2]LIMSI-CNRS
BP 133
F-91403 Orsay Cedex

Abstract

First and foremost, this article presents a conceptual model for ambient computing systems, where we define the vocabulary in use. We then present an architecture that closely matches this model, and that makes use of the popular concept of context component. However, we extend this concept by adding a strong typing of its inputs and outputs, so as to allow easy consistency checks. Moreover, our architecture introduces a high-level mechanism to abstract context and allow the rapid construction of ambient computing applications. At the end of the article, we propose a possible practical implementation of this architecture.

1 Introduction

The goal of ambient computing systems is to interact with users in everyday life, even in situations non typical of human-computer interaction. To this end, these systems must be capable of (1) capturing their *context* of use, and (2) *performing actions* on their environment.

In this article, we mainly focus on the design of an architecture to *capture* context. With "context", we mean what Anind K. Dey describes in his thesis [4]: *"any information that can be used to characterize the situation of an entity"*, where an entity is either a person, a place, or an object.

Thus, entities may be located in the physical world. However, an ambient computing system adds computer objects on top of the physical world. In consequence, user interactions happen in *mixed reality*. For instance, some people have proposed *augmented reality* systems, where interactions happen mainly in the physical world whose physical objects are augmented by virtual properties. Some other people have proposed *augmented virtuality* systems, where interactions happen in a computer-generated world that is augmented by elements taken from the physical world [5].

Context is always a significant parameter, even in *virtual reality* systems that are totally independent of the physical world. In this case, context would be totally inferred from computer representations, with absolutely no link with the physical world.

In fact, Milgram has shown that it is very difficult to precisely define the concepts of *reality*, *augmented reality*, *augmented virtuality* and *virtual reality*. Instead, he introduces a continuum that ranges from pure reality (the physical world) to pure virtuality (virtual worlds): the *reality-virtuality continuum* (fig. 1).

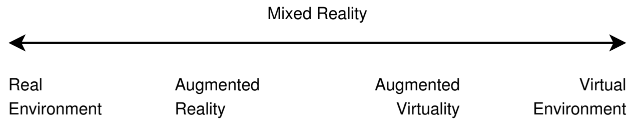

Figure 1: The reality-virtuality continuum ranges from pure reality to pure virtuality.

Frameworks designed to capture context have already been proposed. For instance, Anind K. Dey [4, 3] has released the *Context Toolkit* whose goal is to ease the design and implementation of context-aware systems. Another example is Gaetan Rey's *contextor* abstraction [6]. The contextor is a software component that can aggregate context. We base our work on these previous research achievements. However, our proposal for an ambient-computing platform introduces two original aspects: a strong typing of all communications between software components, and the notion of *object hive*, a software abstraction to ease access to high-level context.

In section 1, we start by defining the underlying vocabulary and our conceptual model for ambient-computing systems. Section 2 details our platform proposal and its original aspects. Section 3 is an overview of a possible implementation for this platform.

2 Conceptual Model

An instance of our conceptual model is presented on figure 2. Abstract representations of objects live in the *model*. They have *incarnations* located in the *world* that can be either physical or virtual.

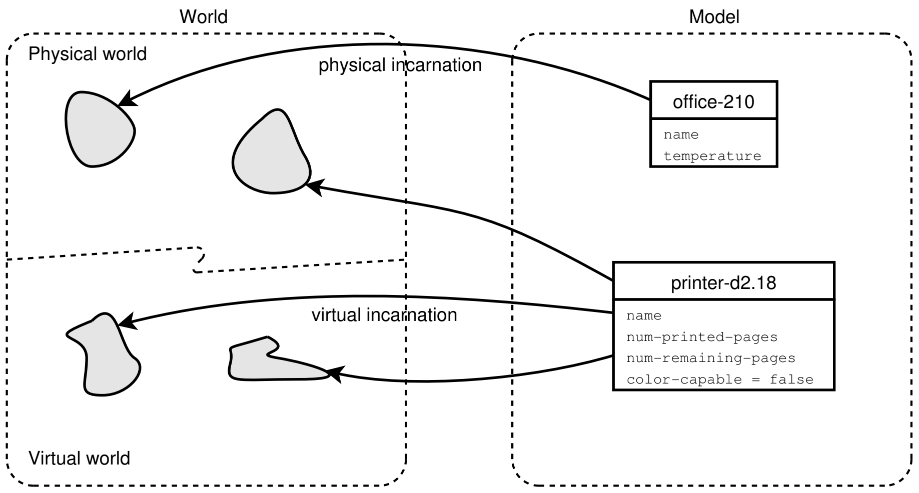

Figure 2: Relationships between objects of the world and objects of the model.

2.1 World

We call *world* the set of all the objects belonging to the reality-virtuality continuum. So the world holds all the objects in interaction with the user, both physical and virtual.

It is nevertheless possible to distinguish between the *physical world* in the one hand, and the *virtual world* on the other hand (fig. 2).

The term *physical world* refers to all the objects that are governed by physical laws. It is the world human beings live in. Conversely, the term *virtual world* refers to environments composed of imaginary computer-generated objects that the user can interact with through virtual and mixed reality applications.

When thinking of virtual objects, one often imagines only images displayed in head-mounted displays, caves and so on. However, virtual worlds can possibly heavily rely on other senses, such as audio and tactile sensations.

In the interior of the reality-virtuality continuum (i.e. in mixed reality), the physical and virtual worlds blend together.

Example 1 — A secretary typesets a letter with a popular word processor. The physical parts of the computer (keyboard, screen, mouse, etc.) are located in the physical world, as well as the secretary herself. Conversely, the letter and the word processor program (as well as the "companion", a small character appearing on the screen and supposed to help the user with her tasks) are located in the virtual world.

However, all these objects belong to the world as a whole. All of them can be sensitively perceived by people.

2.2 Model and Model Objects

We call *model* the abstract representation of the world (physical world as well as virtual world). The objects of the world are described in the model by a set of characteristics called *attributes*.

Each object from the model describes (at least) one object of the world (fig. 2). Since it is impossible to describe every single detail of the world, the model should then be considered a *partial* representation. Indeed, it is likely that implementors would decide to model only the characteristics of the world relevant to the target applications.

It is not possible to automatically check that the model really represents the world it is supposed to describe. This is the ambient environment designer's task to check this kind of consistency. No mechanism can automatically detect possible errors.

Example 2 — In the physical world, a printer is located in room D2.18. It is represented in the model by the object `printer-d2.18`. This object has got an attribute called `printed-pages` that represents the total count of printed pages on the real-world printer located in room D2.18. The ambient environment designer must ensure that this attribute is updated according to its semantics. For instance, if one assigned *another* printer's page counter to it, no formal consistency rule would be broken. Thus, only the environment designer can perform consistency checks because they are not feasible automatically.

There are three types of attributes :

- *static attributes*: values are affected to such attributes once and for all for a given object, and they do not change in its lifetime,
- *state variables*: they represent the dynamic state of the world. They are permanently updated,
- *calculated attributes*: they are dynamically deduced from the values of other attributes. Their value is updated each time one of the attributes they depend on changes.

Example 3 — The attribute `color-capable` of printer `printer-d2.18` is set to `false` for its whole lifetime. Likewise, its `name` attribute is set to `"Printer located in room D2.18"`. A context component regularly queries the printer through SNMP[1] requests so as to know the number of printed pages since the last change of its toner cartridge, and updates the `printed-pages` attribute. The `remaining-pages` attribute is defined to be equal to `toner-cartridge-capacity - printed-pages`. Therefore, it will be updated each time the `printed-pages` attribute is modified.

2.3 Incarnations

We call *incarnation* an object in the physical world or in the virtual world corresponding to an object from the model. In the first case, we call it a *physical incarnation*. In the second case, we call it a *virtual incarnation* (fig. 2).

Example 4 — An incarnation of the `printer-d2.18` object (located in the model) is the (physical) printer located in room D2.18. It is an object from the physical world.

Up to now, we have silently assumed that an object from the model had one and only one incarnation (either physical or virtual). Actually, it is possible for a model object to have *several* incarnations:

1. in *pure reality*, when not interacting with computers, every object has exactly one incarnation, located in the physical world,
2. in *mixed reality*, or simply when interacting with computer tools, some objects of the model can have several incarnations, either physical or virtual,
3. in *pure virtuality*, the objects of the model have one or several virtual incarnations, and no physical incarnation.

[1] Simple Network Management Protocol.

3 Proposal: a Platform for Ambient Computing

3.1 Introduction

The previous section has shown that objects of the world (either physical world or virtual world) can be described in a model by objects whose they are incarnations. Now, we show how this vision can lead to the design of a platform architecture for ambient-computing systems. The general layout of this platform is show on figure 3.

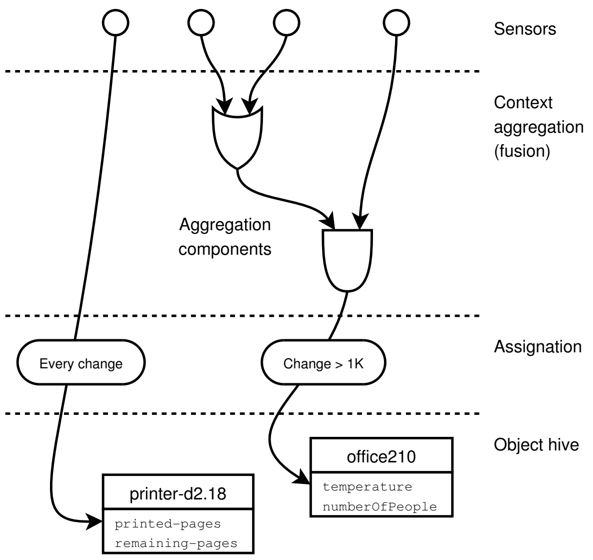

Figure 3: General layout of our ambient-computing platform: context capture subsystem.

The platform is divided in four layers:

Sensors. Sensors are the interface between the world (either physical or virtual) and the platform. They permanently track changes in the world in order to update the model accordingly.

Context aggregation. It is often necessary to combine information from several sensors, so as to deduce relevant and useful context information. That is what we call context aggregation.

Assignation. Information from the preceding layers characterize objects of the model. That is why we *assign* such information to the attributes of the objects of the model.

Object hive. Model objects are gathered in the platform in what we call an *object hive*.

On figure 3, *sensors* and *aggregation components* are designated by the generic term *ambient component*. They share many characteristics with Anind K. Dey's context widgets [4, 3] and Gaetan Rey's *contextors*. The task of these components is to capture information in the world and perform transformations on contextual information.

3.2 Ambient Components

3.2.1 Introduction

An ambient component is a software component that behaves in a relatively autonomous fashion and has got inputs and outputs (fig. 4).

For instance, sensors are particular ambient components that have no input and only outputs. They are representatives in the model for sensors installed in the physical world or information sources located in a virtual world.

More generally, ambient components' outputs are activated or altered in the following two cases:

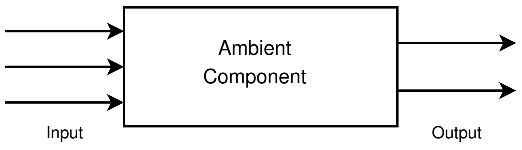

Figure 4: Ambient components.

- when an input changes. In response to this change, the ambient component performs an action, so as to update its internal state as well as its output values. In consequence, the values of some outputs may be modified,
- when an internal *event* happens inside the component, for instance a timeout, or, in the case of a sensor, a change in the world.

Example 5 — The output of a *low-pass filter* component will change every time its input will change. Conversely, we can imagine a *clock* component that will output an event every second. In this case, the component has no input. The cause of output events is totally internal to the component. Likewise, the output of a *thermometer* component embedding a "real" temperature sensor will change depending on the current room temperature. From the platform's point of view, this cause is *internal to the thermometer component* since it has no input.

Ambient components can be interconnected (inside the aggregation layer or the context capture layer, see fig. 3): the output of one component (called *producer* component) is then connected on the input of another component (called *consumer* component). One given output can be connected to an arbitrary number of distinct inputs. However, one given input can be connected to at most *one* output of another context component.

Indeed, when producing information, it is straightforward to distribute it to an arbitrary number of consumers. Conversely, it is very difficult to *fuse* information from several producers to deduce one unique input. It requires (possibly complex) processing that is *specific* to the information involved. That is why an input can be connected to one producer only. However, it is possible that this producer is in fact a fusion component, able to fuse information from several upstream components, each one being connected to one of its own inputs (fig. 5).

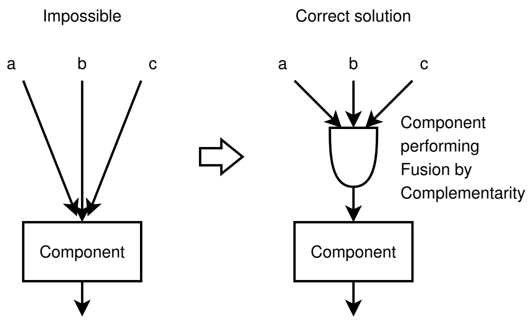

Figure 5: Since one input can be connected to only one output, we must resort to fusion components.

So as to ensure consistency of data processed by the systems, inputs and outputs are typed. Thus, to be able to connect two ambient components, the type of the upstream component's output must be compatible with the type of the downstream component's input. This rule allows the detection of trivial error cases, but does not allow for the detection of more subtle and trickier errors.

Example 6 — On an input supposed to be fed with the acceleration due to gravity (g, measured in $\mathrm{m} \cdot \mathrm{s}^{-2}$ [meters per second per second]), it is *not* possible to connect the output of a temperature sensor (measured in K [kelvins]). However, it *is* possible to connect by mistake the output of a vehicle's acceleration sensor to it, because this quantity is an acceleration too, measured in $\mathrm{m} \cdot \mathrm{s}^{-2}$).

In this example, we have informally shown a first category of typing: typing measured quantities with units from the international system of units (SI). Actually, we propose two categories of inputs/outputs: *value* inputs/outputs, and *event* inputs/outputs.

3.2.2 Value inputs/outputs

Value inputs/outputs can carry two kinds of values:

- *abstract values* (for instance, user identifiers). In this case, we will resort to classical computer types: integers, floats, character strings, structures, etc.,
- *physical quantities* (for instance, a temperature, or an acceleration). In this case, we will give a *unit* (from the international system of units) to inputs and outputs (for example, $\mathrm{m \cdot s^{-1}}$ [meters per second], K [kelvins], etc.)

A value output always has a value. A typical example of context component having a value output is a physical sensor, for instance a temperature sensor. This sensor permanently measures the current temperature, so the value of its output called `temperature` can be read at any moment. A consumer component connected to this output will have several means of retrieving information:

- probing the current output value at a given moment, for instance on initialization,
- subscribing to the producer, and being notified when the quantity fulfills a given condition. For instance: “the absolute temperature change since the last notification is higher than 1 K”, “the temperature has just risen above 273 K”, etc.,
- subscribing to the producer and being notified at a given *sampling* frequency. For instance, a given component can ask to be notified two or three times per second.

3.2.3 Event inputs/outputs

Event inputs/outputs have different semantics. They have no associated value, so one cannot query them at any moment. Conversely, they punctually send messages to the consumers connected downstream. These messages are called *events*. So, the only means for a consumer to connect to an event output is to *subscribe* to this output. This way, the consumer component tells the producer it is interested in the events it produces and wishes to receive them until further notice.

Each event type has got a name that is unique throughout the system. For instance, a crossing detector (such as a *light gate*) sends an event called `crossing-detected` each time someone passes by.

Event types can be classified in a hierarchy, where all events are descendants of a common ancestor, called `generic-event` for instance. This way, an input of one component can be connected to an event output of type T_1 of another component if and only if:

- it is an event input (then, let T_2 be its event type),
- T_1 is a subtype of T_2, ie. an event of type T_1 can be cast to an event of type T_2.

Example 7 — The event type `crossing-detected` described above can have two subtypes, `fast-crossing-detected` and `slow-crossing-detected` (fig. 6). A component that takes in input `crossing-detected` events will also accept `fast-crossing-detected` and `slow-crossing-detected` events.

Example 8 — We can imagine a generic event counter, that would be able to count all occurrences of every possible type of events. To this end, we only need to create a component that has an output of type `generic-event`. Then, it will be possible to connect it to every kind of event output.

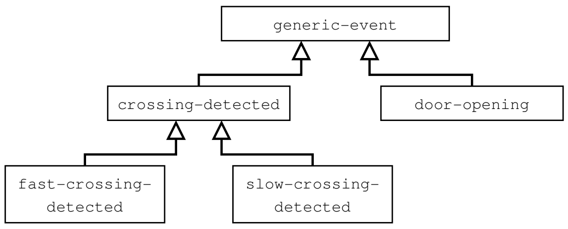

Figure 6: Event type hierarchy.

Some kinds of events are meant to carry information. In consequence, events can have *parameters*, i.e. *attributes*. An attribute has got a name and a type, among value types listed in section 3.2.2: “classical” computer types (character strings, integers, floating point numbers, etc.) or physical quantities with units (for instance, an acceleration in $m \cdot s^{-1}$).

Example 9 — We can define the event type `measured-crossing` that has an attribute called `crossing-time`, that is a physical quantity measured in seconds.

3.3 Context Aggregation

3.3.1 Introduction

The role of aggregation components is to transform raw data acquired by sensors into relevant high-level information about the context of use of the system. As described in [6], this transformation can be performed in several steps. Several layers of aggregation components can transform information step by step.

It is interesting to draw a taxonomy of the different kinds of aggregation components. In this section, we consider ambient components from the point of view of only one of their outputs, which amounts to dealing with ambient components with only one output. This assumption is made without loss of generality, because a component with *n* outputs can be replaced with *n* different components with one output each and all the same inputs as the original component (fig. 7).

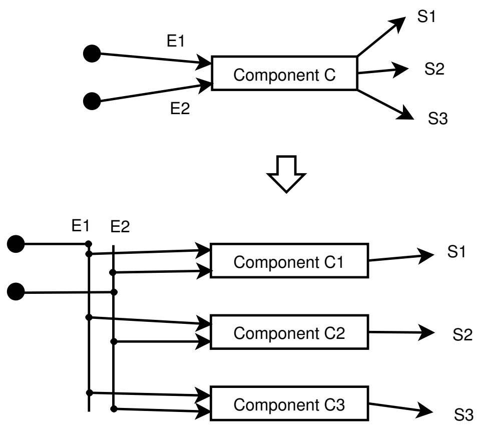

Figure 7: Equivalence between one component with *n* outputs and *n* components with one output each.

First, we distinguish between components with one input and components with several inputs. In the former case, we say that they are *conversion* components; in the latter case, we say that they are *fusion* components.

3.3.2 Conversion Components

A conversion component has got only one input. From values or events in input, it provides other values or events on its output. That is why we say that it performs a *conversion*.

Example 10 — Imagine a component that applies a low-pass filter on its input. It gets values as input, i.e. a signal $f(t)$, and provides other values in output. More precisely, its output is a signal equal to $\frac{1}{T}\int_{t-T}^{t} f(u)du$. The input and the output are of the same nature: they are value inputs/outputs.

Example 11 — Similarly, we can imagine a component that performs a conversion from one event type to another. For instance, one may want to convert from `crossing-detected` events to `person-enters` events if a light barrier is located at the entrance door of a room and therefore detects people coming in.

Example 12 — Imagine a component that detects maxima on its input. It receives values in input but provides events in output: each times it detects a maximum, it sends a `maximum-detected` event. This component's input and output have different natures.

Example 13 — Imagine a component that counts incoming events. Thus, its input is an event input, of type `generic-event`. Each time an event arrives, the component increments a counter, whose *value* is provided in output. Thus, the component's output is a value output. Here too, the component's input and output have different values.

As we see on the example, all combinations of input and output natures are possible. The role of conversion components is *precisely* to perform conversions between all the information types handled by the system.

3.3.3 Fusion Components

A fusion component combines several information sources in input, and merges them to produce a unique output. There are two cases, (a) when inputs provide different kinds of information, and (b) when inputs are meant to provide similar information.

(a) — When its inputs provide different kinds of information, an ambient component deduces different new information. In this case, it is called a *complementarity* component, because it combines its inputs in a complementary way to produce its output. The component performs processing very specific both to its inputs and to the expected result. It is therefore very unlikely to be able to design a generic algorithm capable of fusion arbitrary data in any complementarity component.

Example 14 — Suppose that a system is designed to write the transcripts of meetings. A speech recognition subsystem can provide the raw text of discussions going on, and a video identification subsystem can identify the current speaker. A *specially designed* complementarity component can take in input information from these two subsystems, and provide the complete transcripts of meetings, attributing every statement to the right person.

(b) — When the inputs of one component are meant to provide similar information, we call it an *equivalence* component. For instance, a component can take in input information from three different temperature probes. Several kinds of actions can be performed:

- *redundancy*: in this case, only the inputs that effectively provide values are taken into account, and a "poll" is taken among them. For instance, in the case of three temperature sensors, one can take the *median* value among the three values available. Then, if two sensors out of three provide correct information, the redundancy component provides a correct result, even if the third component provides no or incorrect information. This method is known in the field of system safety as TMR[2]. In this case, it is possible to imagine *generic* redundancy components, that work

[2]Triple Modular Redundancy

on arbitrary data and implement generic redundancy techniques.

- *quality enhancement*: redundancy components only compensate for upstream components' failures. One can imagine more complex processing, that would *improve the quality* of incoming similar information. For instance, by combining information provided by three noisy temperature sensors, it must be possible to reduce the noise in output.

3.3.4 Summary of Aggregation Operations

In this section, we have seen some classes of context aggregation components. They are summarized on figure 8. Our classification is somewhat similar to the taxonomy presented in [6], but our structure is hierarchical and not linear.

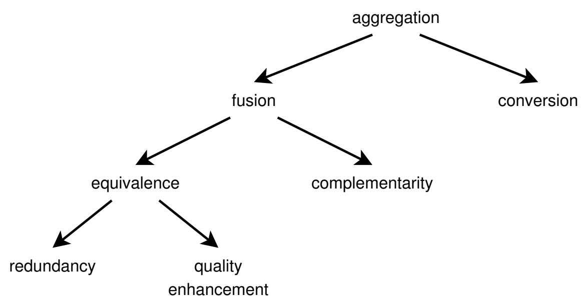

Figure 8: Classes of context aggregation components.

Among these classes, only fusion can possibly be designed to be generic. As for other aggregation techniques, algorithms are very specific:

- to the inputs and outputs of aggregation components,
- and especially, to the *aggregation technique* that one given component implements.

The vocabulary used in this section (equivalence, redundancy, complementarity) is inspired by the research work carried out by Coutaz et al. on the CARE properties of interaction [2]. Coutaz et al. also introduce the notion of *assignation*. We have assignation too, as shown in the following section: in short, assignation allows us to associate the output of context components to the attributes of model objects.

3.4 Object Hive and Assignation

In section 3.1, we have seen that model objects are stored in an object *hive*. The hive is itself an ambient component, that holds a description of every object of interest. Ambient computing applications can subscribe to the hive so as to be notified about changes in the *high level context*. Thus, they receive notifications when model objects' attributes are modified.

Example 15 — an application can subscribe to the `people-count` attribute of the object `office-210` (fig. 3) so as to be kept informed when people enter or leave this office. This is a request on high-level contextual information.

It seems that the *use* of information held in the hive is not problem in itself. However, maintaining the hive in sync with the world is a more complex problem.

Therefore, an update process needs to be performed permanently. This can be done quite simply, in *connecting* the attributes of hive objects to the outputs of aggregation components. This connection is called *assignation*: when the ambient computing system designer decides that an attribute is meant to receive information from a given component's output, he or she somehow *assigns* these information *to* the attribute.

This way, the attributes of hive objects have the same behavior as *consumers* (see section 3.2.1) with respect to the outputs of ambient components they are connected to. In consequence, an attribute can only be assigned a *value* output, because an attribute has got a value at any moment. It would have no meaning to connect it to an *event* output because an event happens only at one point in time.

However, we have seen before that attributes have types, exactly in the same way as value outputs have types. As a result, assignation operations must enforce type compatibility, as when connecting one ambient component to one other. An attribute can only be connected to a *value output of the same type*.

As seen in section 3.2.2, a consumer connected to a value output must subscribe to this output so as to be kept informed when the value changes and meets a given condition. Thus, attributes too must give a condition when subscribing to information providers. These conditions are given when creating the assignation, and are called the *assignation conditions*. For instance, it is possible to subscribe to every change, or only of changes of a minimum amplitude (see the *assignation* layer on figure 3).

Example 16 — The output of the redundancy component described in section 3.3.3 (b), and that combines the outputs of three temperature probes by performing TMR can be assigned to the `temperature` attribute of the `office-210` object located in the hive.

4 Technological Solutions

4.1 Introduction

In this section, we present implementation choices for the conceptual architecture introduced in section 3. The main goals of this implementation are the following:

- the system is distributed across a network; the network is transparent for designers of ambient computing components and applications,
- the system is based on simple and open protocols.

From the description of the model, we deduce that an ambient computing system complying with our architecture is composed of several *ambient components*. Therefore, our proposal is to distribute these ambient components across the computers of a network. Each computer hosts a server, whose task is to enable communication between its own components and other components, either local ones or remote ones.

To support communication between components, it seems reasonable to exchange fragments of RDF[3] graphs. Indeed, RDF is the new standard for the description of semantic information. In addition, vocabulary description functionalities can be added to RDF thanks to related languages such as OWL[4]. This way, it should be possible to describe vocabularies shared by components in a standard manner. For instance, the data type hierarchy could be modeled using OWL.

RDF is a conceptual model, and it has got several representation formats. However, the representation format called RDF/XML[5] seems to be the easiest to use and the most popular, so we have chosen to use it. It is then possible to consider that the servers of an ambient computing system are *web services* that communicate using HTTP[6]. We have resorted to lightweight web services implemented using the XML-RPC[7] protocol, known to be very simple. However, it would be possible to use more complete (yet more complex) web services standards such as SOAP[8].

4.2 Components and Component Identification

As we have shown before, servers are meant to host components and enable communication between them. In particular, the object hive can be considered as one of these components. Therefore, it is hosted

[3]Resource Description Framework.
[4]Web Ontology Language.
[5]RDF over eXtensible Markup Language.
[6]HyperText Transfer Protocol.
[7]XML-Remote Procedure Call.
[8]Simple Object Access Protocol.

on one of the servers of the system, exactly in the same way as any other component is hosted on one server.

Besides, it is useful to have list of all available components at one's disposal, in particular at design time. That is why every server has a particular component called `registry`, capable of providing the list of all components available locally on this server. Registries could even talk with each other so as to build the list of all components available on *the whole network*, and not only locally [1].

Thus, the general architecture of our implementation looks like the example of figure 9. Every server (represented by a 3D rectangle) has got a URI[9] that is made of the physical (IP) address of the server and its port number. Within a given server, every component has got a unique name. There are some special components, such as the `registry` component, available on every server, and the `hive` component, located on *one of* the servers.

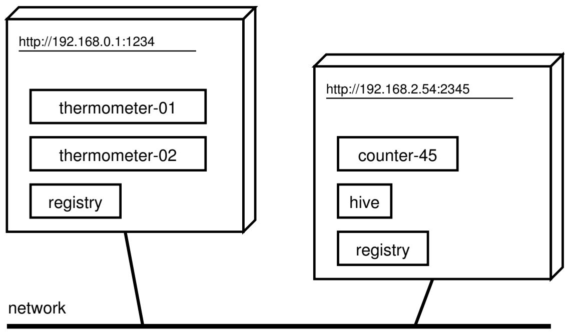

Figure 9: Example of deployment of an ambient computing system.

This way, it is possible to define a URI for each component of a system, simply by concatening the server URI and the local name of the component, separated by a slash.

Example 17 — The `thermometer-02` component, located on the server identified by the URI `http://192.168.0.1:1234`, is itself identified by the following URI: `http://192.168.0.1:1234-/thermometer-02`.

As a result, it is easy to name any component by its URI, which is very important when using RDF because all RDF objects (called *resources* in the dedicated vocabulary) *must* be identified by URIs. In our system indeed, URIs are *both logical* identifiers (with respect to RDF) *and physical* identifiers, enabling access to platform components through the layers of a network.

5 Conclusions and Perspectives

In this article, we have first presented the vocabulary for a conceptual model of ambient computing systems. We have then proposed an architecture model for ambient systems that fits this conceptual model. This architecture model resorts to aggregation components that have already been presented by others.

However, our model introduces a strong typing of the messages exchanged between ambient components. Besides, the notion of *hive* constitutes a high-level yet easy-to-use abstraction to access context information.

Some context platforms such as the Context Toolkit [4] deal with the storage of context history and synchronization of context components. We have not yet introduced such facilities in our architecture for the moment because they are not our main topic of interest, but we may add some of them in the future.

Our short term research directions are the finalization of the test implementation, improvements to the type system, and the possibility to provide detailed semantic information about context components.

[9]Uniform Resource Identifier

References

[1] C. Bettstetter and C. Renner. A comparison of service discovery protocols and implementation of the service location protocol. In *Proc. EUNICE Open European Summer School*, Twente, Netherlands, Sept. 2000.

[2] J. Coutaz, L. Nigay, D. Salber, A. Blandford, J. May, and R. M. Young. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. In *Proceedings of INTERACT'95: Fifth IFIP Conference on Human-Computer Interaction*, pages 115–120, 1995.

[3] A. K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[4] A. K. Dey and G. D. Abowd. *Providing architectural support for building context-aware applications*. PhD thesis, Georgia Institute of Technology, 2000.

[5] L. Nigay, E. Dubois, and J. Troccaz. Compatibility and continuity in augmented reality systems. In *I3 Spring Days Workshop, Continuity in Future Computing Systems*, Porto, Portugal, Apr. 2001.

[6] G. Rey and J. Coutaz. Foundations for a theory of contextors. In *Computer-Aided Design of User Interfaces III*, pages 13–32. Kluwer Academic Publishing, 2002.