O3MiSCID, a Middleware for Pervasive Environments

Rémi Emonet, Dominique Vaufreydaz, Patrick Reignier, Julien Letessier

O3MiSCID, a Middleware for Pervasive Environments

Rémi Emonet, Dominique Vaufreydaz, Patrick Reignier, Julien Letessier
PRIMA - INRIA Rhône-Alpes
Zirst, 655 avenue de d'Europe, Montbonnot, 38334 Saint Ismier cedex, France
{remi.emonet,dominique.vaufreydaz,patrick.reignier,julien.letessier}@inrialpes.fr

***Abstract*— This paper introduces a new lightweight middleware for pervasive environments. This middleware abstracts network communications and provides service introspection and discovery using DNS-SD (*DNS-based Service Discovery* [1]). Services can declare simplex or duplex communication channels and variables. The middleware supports the low-latency, high-bandwidth communications required in interactive perceptual applications. It has been designed to be easy to learn in order to stimulate software reuse in research teams and is revealing to have a high adoption rate.**

I. Introduction

The fast emergence of pervasive computing is transforming the way the applications are designed. Applications are now commonly distributed on numerous computers with various hardware capabilities (from high-end servers to PDAs). Building mobile, distributed and robust applications is a challenge for system designers and developers. To cope with this new complexity, some pieces of middleware are being designed. They help the designers and developers by abstracting the lower-level details of application development and providing higher-level concepts to manipulate.

The concept of zero-configuration networking proposes the following approach: all application parts are services that are declared over the network. When a service requires another to function, it doesn't need to known its network location (host and port): a service discovery mechanism can provide it, based on its type and its properties (a discovery query). The same approach is used when browsing the yellow pages to find a plumb: you know what service you need but don't know yet the exact coordinates where to contact its provider.

All service-oriented middleware provide service lookup; however, some of them require the application to know where to find a centralized service repository. There are mainly two existing infrastructures for zero-configuration networking that do not rely on a centralized well-located service repository: UPnP and Zeroconf.

UPnP (*Universal Plug and Play* [2]) exposes a large set of features including the SSDP (*Simple Service Discovery Protocol*) protocol that handles service discovery. Although UPnP is widespread, very attractive and fulfills many of our requirements, it has a drawback that is redhibitory for the use we want to make of it. UPnP does not have a mechanism for instant notification when a service becomes inaccessible, and thus, there is no reliable way to have an up-to-date view of the available services.

The second major infrastructure for zero-configuration networking is Zeroconf [3]. Zeroconf leverages DNS-SD [1] combined with Multicast DNS [4] to achieve distributed service discovery with quick service connection and disconnection notifications.

There are also some dedicated middlewares such as [5]–[8], but none is suitable to our requirements. Although some are designed with context-aware applications in mind, which is one of our goals, most of these middlewares are too specialized or Java-centric, or both. None of these middlewares fulfills all our needs; in particular user-friendliness. We choose to propose an easy to learn middleware based on DNS-SD.

In section II we will first identify the requirements for a user-friendly, platform-agnostic middleware for interactive applications. We then present our solution, O3MiSCID (section III), the Object-Oriented Opensource Middleware for Services Communication, Introspection and Discovery. Two examples using this middleware are detailed in section V. Finally we will conclude and present the future work related to this middleware.

II. Objectives and Requirements

In this section we detail the requirements our middleware should meet: attractiveness, network efficiency, robustness, ease of configuration and extensibility.

1) Attractiveness and availability: The main goal of a middleware is to allow (independently developed) software components to be uniformly packaged as services and cooperate. The middleware should be sufficiently attractive: most reaserchers and developers should feel that the benefit of using it will quickly outweight the development overhead. Moreover, developers have different backgrounds, interests and skills. The middleware must be available to any potential user: it must be cross-language, cross-platform, and easy to learn for anyone.

2) Network efficiency: Our middleware needs to be a general purpose middleware, with a specific constraint: it must be useable in the context of interactive applications involving audio and video processing. To match the interactive constraints, it must keep a low latency in the communications. Perceptual processes such as audio-video analysis require high throughput: the middleware should not overload the communication cost between services.

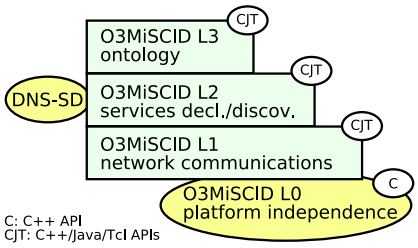

Fig. 1. The layered architecture of the proposed middleware
Each layer is accessible using dedicated APIs currently available in Java, C++ and Tcl.

3) Robustness: We split this requirement in two categories: middleware robustness and service robustness. Regarding the former, we note that if a centralized middleware approach is used, e.g. with a single middleware server offering services to a network of computers, the failure of the server may cause the failure of all client applications. Application reliability requires middleware robustness, which in turn imposes to rely on a decentralized approach.

In pervasive applications, services are often dynamically added or removed. Services should thus be robust to communication loss, and have easy access to up-to-date service availability information. They can then be designed to recover from disconnections by using the list of available services provided by the middleware.

4) Ease of configuration: Distributed software consists in splitting a complex application into multiple processes (services), usually spread on multiple computers. Configuring such software proves challenging, as services may need to find other dependant services to function. In order to easily deploy an application in a new site, we reduce the number of configuration files specifying the interconnections by relying heavily on auto-detection features. It is the task of service designers to ensure strong autonomy in software configuration, however the middleware must make this task as easy as possible.

5) Extensibility: The middleware must be easily extensible: one should be able to design dedicated frameworks featuring higher abstraction level concepts. Such an extension mechanism must fit into the existing middleware. Some of the possible extensions we currently envision are a context awareness extension and an extension to help in designing "autonomic systems" [9].

III. Middleware Architecture

In this section, we present the middleware architecture we propose. It is layered (see fig. 1): layer 1 handles and abstracts communication mechanisms; layer 2 handles services description, discovery and introspection; finally, layer 3 manages ontological manipulations of the services.

A. Layer 1: Network Communications

The lowest layer features a lightweight communication protocol and programmatic facilities to handle communications. We use the BIP communication protocol (*Basic Interconnection Protocol*, [10]), which splits the communication stream into messages. This is achieved by adding a 34-byte header to each message to be sent. This lightweight header contains, among other information, the size of the message payload. This content can contain arbitrary data such as a binary stream, ASCII text, or an XML fragment. Given the small size of the header and the absence of limitation on the message content, this protocol is sufficiently general purpose and can handle high bandwidth messaging and streaming.

Any application using the proposed protocol is called a BIP peer. All services (see next section) are BIP peers but not all BIP peers are services. In particular, any peer can use BIP to access running services without having itself been declared as a service. In other words, layer 1 can be used independently, which allows for easy interoperation.

B. Layer 2: Services

The service layer is built on top of the communication layer and is dedicated to service description, introspection and discovery. Basically, a service is described by its name, its input/output communication channels and other service-specific parameters. This subsection details the service-related concepts manipulated by O3MiSCID.

1) Service Description and Introspection: A service must have a unique identifier: a human-readable name that is unique, network-wide. A service can expose a set of channels. Each channel is a communication "port" that can be mono- or bi-directional channel (*input*, *output*, or *inoutput* channels). In the scope of a particular service, each channel has a unique name, a human-readable description string, and a type string that describes the data format used inside this channel. A service can also export state parameters called variables. Variables can be read and written depending on access rights set to it. Each variable also features a type and a description. A peer can subscribe to variable modifications and receive notifications each time the variable value is changed.

Each service has at least an *inoutput* communication channel called the control channel, dedicated to introspection queries. This allows to list the channels and the variables exported by the service. It is also used to interact with the service: query, update, and subscribe to variable changes.

2) Service Discovery: The middleware allows a client to register new network services and provides two ways to discover available services. A service browser can be notified whenever a service event occurs (service appearance or disappearance). Alternatively, a client can use a wait-for-service functionnality: one provides a service description, and is notified as soon as a matching service is present. Such descriptions are *service filters*: conditions that can be based on any service attribute such as service name or host; channel name, description, format, or supported versions; or service variable.

3) User Point of View: The exact use of the middleware may differ slightly depending on the programming language that is used. However, using this middleware basically consists

in instantiating classes representing a service to register. Then, one can define inputs and outputs (directly in the source code or through an external XML description file, see below), and eventually bind input channels to callback functions. Similar actions are possible for the service variables.

The middleware provide also all the required facilities to inspect and discover remote services. Information about the remote services is accessible programmatically. Filters for the discovery process are manipulated as functors (objects representing a function); in our case, a boolean function taking a service description as a parameter.

C. Layer 3: Ontology

The ontology layer is the highest abstraction layer of O3MiSCID. For the moment, it is only preliminary and the object of current studies. Choices were made in order to check the potentiality of the method. They are not definitive.

There are two requirements driving the ontological layer design. From a system-centric perspective, we need to store information about "the world", i.e. everything related to the perceptive environment, including any hardware and software. This data will be used for monitoring (see section V-A) and service selection purposes for example. The main system issues are information sharing, availability and consistency. From a user's point of view, we aim to provide a means to define a service easily. The end user should only write a terse description of a service to launch it. Another key point is how a service can be found using not only its name as a query. Users will be reluctant to write a programmatic service description to discover a suitable service.

Concerning the user standpoint, we need to provide a means to describe the world. We experimented with two languages: the OWL ontology language and a custom XML language. OWL is currently used in many pervasive ontologies [11] [12]. Even if it is an expressive and powerful language, it is not well known and it requires us to add a reasoner to our middleware. Furthermore, developers are generally proficient in XML-based technologies, and O3MiSCID already uses XML (for the service inspection interface). Our current choice is thus XML.

Using XML, the user can now only write a concise description of the service like in the following example code:

```
<Microphone freqMin='50' freqMax='10000'
    type='long' id='MicroLong3'
    sensibility='5'>
  <position3D X='1.78' Y='5.75' Z='1.38' />
  <orientation horizontal='180' vertical='0'
    axial='0' />
  <genealogy>
    <branch>Hardware::Microphones</branch>
  </genealogy>
</Microphone>
```

This XML code is used to define a microphone service. The end user calls a class constructor with it or a file containing the code to publish the service. Channels and variables are already set with values found in the XML description. By generating multiple descriptions, one can declare multiple services simultaneously and obtain an array of ready-to-use services.

Another application of using such descriptions is the ability of a service to register to several services in order to get a partial vision of the world. For example, a service that needs to use cameras can register for all cameras (current and future) in order to dynamically receive their XML description. It can then build an XML tree using this information and query it using XPath to find the best suitable camera. Creating a partial vision of the world and querying over the resulting XML tree represents only 2 method calls.

IV. O3MiSCID Implementation

In order to cope with our requirements, we need to make several technical choices for the implementation. In this section, we detail and justify these choices, while presenting an overview of O3MiSCID's programmer interface.

A. Conformance to layer requirements

1) Layer 1: The implementation of the network communication layer provide programmatic facilities for communication. Classes are provided to create client, server or bidirectional client-server socket objects that communicate using the BIP protocol. Once instantiated, client objects can be used to send messages; a callback function is called on the server for each message received. Other facilities are provided such as message interpretation (as byte array for binary messages, as string for plain text, as DOM tree for XML documents...) and listing of the clients connected to a server.

To ensure the reliability of the communications, TCP-only connections are used by default by the middleware. However, for some applications, latency is a more important constraint. To cope with this, the middleware allows also mixed TCP/UDP connections to be used. Indeed, tightly coupled interactive applications [13] implemented using this middleware meet the typical human-computer interaction latency constraint (50 ms).

This lightweight lower layer is responsible for meeting the performance requirements.

2) Layer 2: O3MiSCID handles service discovery and registration using a DNS-SD infrastructure. The mainstream DNS-SD implementation (branded *Bonjour*) uses multicast DNS (mDNS) to advertise all the existing services in a non-centralized fashion: there is no central service repository. This allow to meet the robustness requirement. Available mDNS toolkits provide a daemon running on each machine. When an application wants to register a service or browse the available services, it addresses its queries to this daemon, via a dedicated API. An additional advantage provided by DNS-SD is that any system is instantly notified of service failure or disconnections. This property is mandatory for reactive networks to be able to maintain an up-to-date list of available services.

Given the presented concepts of channels, control channel, variables, and introspection, O3MiSCID uses DNS-SD to simplify the implementation of some of the concepts. The

DNS SRV record is used to store the name of the service, the machine it is hosted on and the port of the control channel of the service. The TXT record having a limited size, it is used as as a cache for static service information that is also available through control queries. The main information cached in the TXT record are the list of input, output and inoutput channels, the TCP and/or UDP port number associated to the channels and an information string designing the owner of the service. This permits for performant query-based discovery of services, as it allows to shortcut service inspection for simple queries, and to accelerate more complex queries.

3) Layer 3: In order to cope with the system requirement of the third layer, we need to choose an architecture. Most of the current systems (e.g. [11]) use a centralized approach. On one hand, it allows easy sharing of ontological data with low protocol overhead. On the other, in the event of a server failure, all information about the environment is rendered unavailable. Inconsistencies may also arise because of the network latency between changes in a service's perception of the world and the availability of this information on the server. In other words, asking information about a service at time *t+1* may provide outdated information because the state of that service at time *t* is only updated at time *t+2* on the server. The problem is the same for services that crash, or simply stop.

For all these reasons, using a distributed architecture seems suitable. The first idea that comes to mind is to use several ontology servers. It solves the "server crash" issue but not the data consistency problem: replication and propagation of information is a complex problem we do not want to address. Thus, we decided to delegate information sharing to the services themselves, in a peer-to-peer fashion. If one needs information about a service, it is simply queried. If up-to-date information is required, one may also register to be notified, and receive messages if the service state is modified. The consistency problem then disappears, as any service only sends its current state. Moreover, if a service crashes, DNS-SD notifies all interested peers, and other services will stop querying.

B. Multiplateform cross-language implementation

One of the design goals of this middleware is to be simple enough to be easily implemented. This goal has been achieved, as a few implementations are already available in various languages ranging from an interpreted scripting language (Tcl), to a high performance language (C++), and to a high productivity language (Java).

On top of the basic full featured Java version of the middleware, an OSGI version has also been built. The OSGI packages in the Java version expose a higher abstraction level interface that hides the technical details of the middleware.

The C++ version of the middleware is fully crossplatform and works on Windows, Linux and MacOSX. To achieve this ability to run on any platform, an abstract layer has been added. This layer abstracts the system-level base objects such as threads, mutexes and sockets (see fig. 1).

Depending on the needs, further implementations may be written. By now some interesting languages and implementation we can think of are Python and C# (to cleanly integrate with the .NET framework and all its languages).

V. Example Applications

In this section we give a simplified illustration of how the middleware can be used. The examples we present illustrate the usage of different kinds of services at different levels of abstraction: hardware services (microphones, cameras), perceptual services (speech activity detector, visual trackers) and a few higher-level services like a context modeler or a dialog manager.

Our field tests take place in a smartroom environment. This room is equipped with multiple cameras, microphones and is used to run many O3MiSCID services. We will first show that our middleware can bring us facilities for monitoring hardware, software and even human activities in this room. We will next present a multimodal perceptive application designed to automatically record seminars and conferences for instance.

A. Perceptive environment and services monitoring

When working in a perceptive environment, monitoring issues are critical. There are many things we need to observe. We may need to keep an eye on the perceptive environment itself, on the currently running services and their status, or results from this services, for instance the location objects detected by a tracking service. In our approach, all this information (what we name "the world"), is described by a set of services. Indeed, as seen in section III-C, each service provides information about itself by answering to queries over its control channel.

In the following example, the world is our smartroom with sets of microphones, cameras, beamers, trackers, a speech activity detector, a visual tracker, a movie director, etc. In this case, we can extract much information for distributed data over the O3MiSCID middleware by browsing existing services and instrospecting them (see Section III-B). First, we can dynamically extract information about the room and automatically construct a 3D representation. On figure 2, we can see the actual position and orientation of cameras (around the room ceiling), the array of microphones (on the wall on the top of the schema) and a steerable camera-projector pair [13]. We can also display on-line changes like the movements of mobile pan/tilt cameras, by registering to the orientation variable of the corresponding camera services; or display targets found a visual person tracker, appearing and moving inside the room.

Using the introspection capabilities of O3MiSCID, we can also obtain the interconnection graph of all running services in our environment (Figure 3). In this example, the connectors do not presume of the direction of data between services. We can see different types of connections between services:

- the first type concerns standalone services (BlueTooth-Scanner, AudioRouter...). They do not have any client connected to them at the graph construction time;

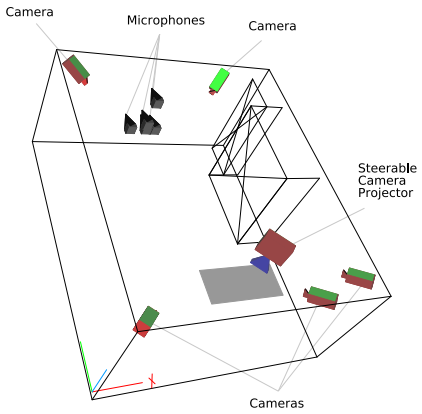

Fig. 2. Gathering and representing data from O3MiSCID

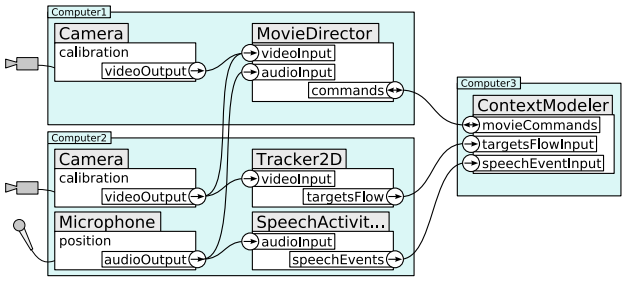

Fig. 4. Example of the "Automatic Cameraman" distributed on 3 different computers. Input, output and inoutput channels of the services are represented by the circles. Curves are representing the connection between the channels. Note that the control channels of the services are not shown in this figure.

- next, we find a group of interconnected services. All of these connections have been determined fully automatically. This group of services constitutes the Automatic Cameraman and it will be detailed in the next section. It is a typical example of a perceptive application composed of many services, as targeted by our middleware, thus a good case study;
- the last type that we can see is the connection between the `66cd4567` client and the `out3DTracker` service. Indeed, as already mentioned in section III-A, one may connect directly to a service using the BIP protocol [10] using only the first O3MiSCID layer (or even without using O3MiSCID).

To conclude this section, our middleware gives multiple ways to observe the pervasive environment from the system, the application, or the user perspective. The introspection and distributed descriptions mechanisms offer many monitoring facilities. It is easy to envision number of applications still allowing to enrich these capabilities.

B. Automatic cameraman

The automatic cameraman is a multi-services application able to determine what happens in a room equipped with microphones and cameras in order to record the most representative movie. It has been successfully field tested to record a full conference [14].

Our virtual test case takes place in a room containing two cameras and one microphone plugged on two different computers. Services associated to the hardware are started on two machines to handle the data acquisition process. Camera services have a calibration variable (a 4x4 matrix encoding the position, orientation and internal parameters of the camera) and an output channel where each frame is written. The microphone service is quite the same: it has a position variable and an output channel.

Figure 4 shows perceptual services and two higher level services. A visual 2D tracker service is responsible for tracking people into the room using a camera and a homography (this tracker and more generally internal processing of services are not in the scope of this article.). A speech activity detector uses a microphone and determines if there is someone speaking in the room. The context modeler takes outputs from the two previous services and performs reasoning to understand what is going on in the room (who is here, where are they, and what are they doing). It outputs commands (change camera view or move mobile camera for example) to the Movie Director service, which is responsible for the production and/or streaming of the final movie.

Actually, the Automatic Cameraman tends to construct this kind of movies using several microphones and cameras. To end up with the connection graph pictured in figure 4, the Movie Director service, for example, first queries a list of all potential services and performs introspection on them using Layer 2 (see III-B). It lists all the cameras and retrieves their position (extracted from their calibration matrix). It does the same with all the microphones. Then it selects the microphone service and the camera service that are closest to each other and then connects to them. We can note that, in this case, the service is doing much more work (positions extractions and distances computations) than just expressing a request describing its needs.

VI. Conclusion and Future Plans

In this paper, we presented O3MiSCID: an Object Oriented Opensource Middleware for Service Connection, Introspection and Discovery. This middleware has a layered architecture. The lowest layer abstracts network communications using BIP/1.0 [10], a performant, low-overhead protocol. The second layer is dedicated to creation, introspection and discovery of BIP services. The last layer, which addresses ontology and more advanced features, is still under development. This middleware is specifically designed for pervasive environments and can maintain low latency and high bandwidth communications, allowing the creation of context-aware interactive applications featuring for instance audio stream and video stream processing.

Its implementation is based on DNS-SD (DNS Service Discovery) over mDNS (multicast DNS) and features a robust decentralized architecture for service advertising and discovery. DNS-SD we favored over UPnP because of the requirement of have highly coupled interaction between services and so require quick and robust service notifications of service status

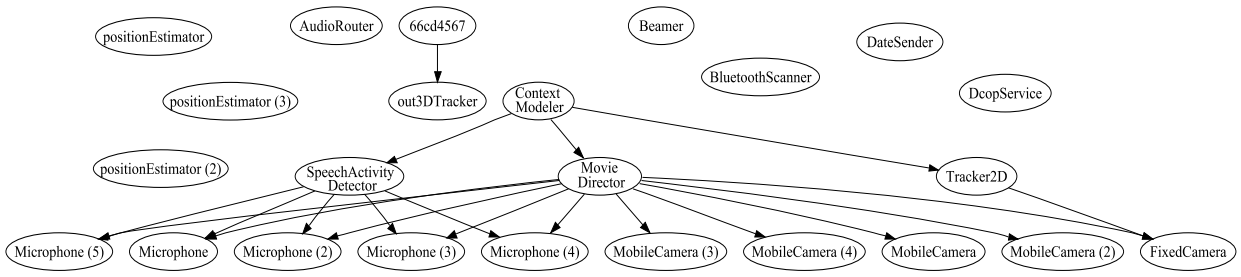

Fig. 3. Interconnection graph of services running in our perceptive environment

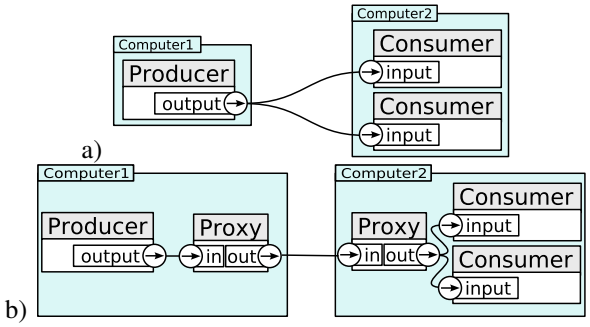

Fig. 5. Proxy Architecture
a) Two clients on a same machine are connected to a same remote service.
b) The addition of a proxy services result in dividing by two the network load. If a proxy is not present or down, the default behavior can still be used as a fallback.

updates, in particular connection and disconnection. DNS-SD also provides a convenient built-in handling of unique names and limited service descriptions.

Implementations already exist in multiple languages such as Java, C++ and Tcl. This middleware is already used by three research teams to deploy interactive applications prototypes, and is revealing to have a high adoption rate.

Since we aim at a high quality, open source release of the O3MiSCID middleware, we still have to achieve cross-implementation code review and interoperation tests to ensure the coherence between all the implementations. New documentation and tutorials also need to be written. We are currently working on this point in order to quickly distribute O3MiSCID on the Sourceforge platform or similar.

A general proxy architecture would be an interesting addition, in order to optimize the network load. Figure VI gives an illustration of the proxy approach. This proxy system is particularly needed for wide streams such as audio and video streams; some middlewares are in fact dedicated to such proxying [15].

The middleware currently satisfies all needs concerning discovery and introspection of the existing services. Through enhancing ontological information about the services and their operations, we envision to tackle with the problem of automatic service configuration and composition.

ACKNOWLEDGMENT

The authors would especially like to thank Sébastien Pesnel, former PRIMA project member, for his considerable development effort concerning the O3MiSCID middleware.

REFERENCES

[1] DNS-SD web site http://www.dns-sd.org/. [Online]. Available: http://www.dns-sd.org/
[2] "UPnP Device Architecture," http://www.upnp.org-/download/UPnPDA10_20000613.htm, 2000.
[3] E. Guttman, "Autoconfiguration for IP Networking: Enabling Local Communication," *IEEE Internet Computing*, pp. 81–86, June 2001.
[4] Multicast DNS web site http://www.multicastdns.org/. [Online]. Available: http://www.multicastdns.org/
[5] T. Gu, H. K. Pung, and D. Q. Zhang, "A service-oriented middleware for building context-aware services," *J. Netw. Comput. Appl.*, vol. 28, no. 1, pp. 1–18, 2005.
[6] H. Chen, T. Finin, and A. Joshi, "Semantic web in the context broker architecture." [Online]. Available: citeseer.ist.psu.edu/646646.html
[7] M.-T. Tran, B. Hirsbrunner, and M. Courant, "A context-aware middleware for multimodal dialogue applications with context tracing," in *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*. New York, NY, USA: ACM Press, 2005, pp. 1–8.
[8] Jini web site http://wwws.sun.com/software/jini/. [Online]. Available: http://wwws.sun.com/software/jini/
[9] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
[10] J. Letessier and D. Vaufreydaz, "Draft spec : Bip/1.0 – a basic interconnection protocol for event flow services," 2005. [Online]. Available: http://www-prima.imag.fr/prima/pub/Publications/2005/LV05/
[11] A. Paar, J. Reuter, and J. Schaeffer, "A pluggable architectural model and a formally specified programming language independent api for an ontological knowledge base server," in *Australasian Ontology Workshop (AOW 2005)*, Dec. 2005, publikation.
[12] H. Chen, F. Perich, T. Finin, and A. Joshi, "SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications," in *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, Boston, MA, August 2004.
[13] S. Borkowski, J. Letessier, and J. L. Crowley, "Spatial control of interactive surfaces in an augmented environment," in *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Bastide, P. A. Palanque, and J. Roth, Eds., vol. 3425. Springer, jul 2004, pp. 228–244, eHCI 04. [Online]. Available: http://www-prima.imag.fr/prima/pub/Publications/2004/BLC04/
[14] F. Metze, P. Gieselmann, H. Holzapfel, T. Kluge, M. Wolfel, J. L. Crowley, P. Reignier, D. Vaufreydaz, F. Bérard, B. Cohen, J. Coutaz, S. Rouillard, V. Arranz, and M. Bertran, "The fame interactive space," in *2nd Joint Workshop on Multimodal Interaction and Related Machine Learning Algorithms*, Edinburgh - UK, feb 2005, p. 4. [Online]. Available: http://www-prima.imag.fr/prima/pub/Publications/2005/MGHKWCRVBCCRAB05/
[15] NIST Smart Data Flow System web site http://www.nist.gov/smartspace/nsfs.html. [Online]. Available: http://www.nist.gov/smartspace/nsfs.html