The Contextor Infrastructure for Context-Aware Computing

Gaëtan Rey, Joëlle Coutaz
IIHM, CLIPS-IMAG, 385 Rue de la Bibliothèque,
38240 Grenoble Cedex 9, France
{Gaetan.Rey, Joelle.Coutaz, iihm}@imag.fr
http://clips.imag.fr/iihm

Abstract. This article is concerned with the development of software infrastructures for context-aware computing. To clarify the diversity of approaches in this area, we propose a synthesis of the state of the art using a reference framework derived from sound software engineering design principles. We then present our own contribution, the Contextor Infrastructure. We describe its conceptual computational model and provide implementation details.

1 Introduction

Context is an old friend. Since the early sixties, it has been modelled and exploited in many ways in the areas of operating systems, linguistics, AI, and HCI. In ubiquitous computing, the scientific community has rediscovered this concept and has debated for more than five years without reaching a consensus for an operational definition. It is however admitted that: 1) Context does not exist out of context: it is defined in relation to a purpose; 2) Context is an information space that serves interpretation [1]; 3) Context evolves, is structured, and shared. The working hypothesis is that context-aware computing is key to the emergence of new forms of services that can be intertwined with human activities at multiple scales. Because of the lack of predictive theories for prescribing the appropriate use of context, the research community has adopted an empirical approach with the development of small-scale concept demonstrators. As for many research areas, this prospective phase has generated the need for new tools to explore the problem space and deploy full-scale solutions.

In this article, we are concerned with the software tools that support the iterative and incremental development of context-aware systems. The Context Toolkit (CTK) is the seminal example in this area [2]. Other tools such as [3,4,5,6,7], address similar issues. To clarify the diversity of approaches, we propose a synthesis of the state of the art using a reference framework derived from software engineering design principles. We then present our own contribution, the Contextor Infrastructure, from its theoretical foundation to its conceptual architecture and operational implementation.

2 Synthesis of the State of the Art

The problem is to provide developers of context-aware systems with software building blocks that support their programming task in an effective manner. Like most conventional systems, context-aware computing is concerned with heterogeneity, distribution, interoperability, extensibility, and functional decomposition. Unlike conventional systems, context-aware computing has to support these aspects at multiple scales, from the ultra-small to the planet-level, all of this exacerbated by the spontaneous occurrence of unpredicted phenomena such as the migration, reconfiguration, uncertainty, maleficence, and failure of the physical world (humans, and artefacts). In this section, we investigate how these aspects can be addressed reusing software engineering design concepts and principles. In the light of these requirements and recommendations, we analyse key examples of development tools from the state of the art in context-aware computing. Table 1 synthesizes our survey.

2.1 Requirements and Recommendations

Heterogeneity. In software engineering, heterogeneity is addressed by means of encapsulation. Encapsulation creates an abstract barrier that hides the low level functioning of an entity and defines an interface through which the entity can be used. However, in context-aware computing, adaptation is needed to cope with many forms of spontaneity and uncertainty. Therefore, the entity should also be inspectable and controllable. Reflection provides this capability through meta-level interfaces [8]. In particular, meta-level interfaces can be

used to express the quality of service expected from a particular component, as well as the quality of the results provided by this component. In context-aware computing, data is not necessarily certain. Confidence factors used in AI, as well as metrics such as latency, precision, stability, and resolution, can be usefully exploited to express QoS through meta-level interfaces. As shown in Table 1, very few tools support both meta-data as well as inspection and control.

Distribution and interoperability. Network technologies and standard communication protocols are key to distribution and interoperability. Recent developments in ad hoc networks offer potential solutions to the fundamental multi-scale spontaneity nature of context-aware computing. Wireless hosts can communicate with each other in the absence of a fixed infrastructure. An ad hoc network forms a 'connectivity island' whose horizon can evolve dynamically: it can detect new comers and leavers; it can split into multiple islands and two islands may merge dynamically. Therefore, tools for context-aware computing should draw upon network services for topology control, message routing, broadcasting, multicasting, and geocasting. However, we must remember that the very existence of 'connectivity islands' implies that the discovery and use of context-oriented services cannot rely exclusively on the fixed planet-level infrastructure. Therefore, a P2P model, rather than a conventional client-server approach, should be favoured to support distribution.

Table 1. Survey of infrastructures for context-aware computing. The 8th column corresponds to our own contribution.

	CTK [2]	**IrisNet [3]**	**Confab [4]**	**SCI [5]**	**Aura (CIS) [6]**	**Gaia Context [7]**	**Contextor Infra-structure**
Levels of abstraction	. Sensing . Transfor. . Service	. Sensing . Service	. Service	. Sensing . Transfor. . Service	.Service	. Sensing . Transfor. . Service	. Sensing . Transformation . Service
Comput. Model	. Process . Hybrid (situation abstract.)	. Data	. Data (tuples)	. Process	. Data	. Data (1rst ord. logic)	. Process (sensing&Trans levels) . Hybrid (other levels)
Discovery mechanism	Yes (server)	Yes (Orga-nizing Agent)	Yes (Explicit connect.)	Yes (Context Utilities)	Yes (Explicit connec.)	Yes (server)	Yes (no server)
Network coverage	Local	Global	Local & Global	Local & Global	Local & Global	Local	Embedded, Local & Global
Self.Config	No (except for the situation. Abstract.)	Yes	Yes	Yes	Yes	Yes	Yes
Data acquisition	Subs-Notif.	Req-Answ	Req-Answ	.Req-Answ .Subs-notif	Req-Ans	.Req-An .Subs-not	.Req-Answ .Subs-Notif
Meta-interface	No (but planned with enactors)	Yes	Yes (tuple level)	Yes (Profile request)	Yes (through meta-data)	No	Yes (Sensing &Transform. levels)
Meta-data	No	No	Poor (CF only)	Yes	Yes	No (but planned)	Yes
Orthogo-nal ser-vices	. History	. History . Privacy (sensing level)	. History . Privacy (access control to tuples)		. History		. Privacy at the sensing and transformation levels via login password
Existence of servers	Yes (discov-erer, history)	Yes	Yes (Infospace servers)	Yes (Context Server)	Yes	Yes	No

On top of the network infrastructure, communication protocols for context-aware computing need to cope with an evolving set of data types whose instances correspond to possibly hypothetic phenomena: as stated in the introduction, context is ill-defined, unbounded, built from a variety of data sets. Therefore, communication protocols for context-aware computing should 1) define or reuse standard communication protocols based on XML-self-describing data, and 2) propose two approaches to data acquisition: the query-answer method for applications that request for a precise thing right now on the spot, and the subscribe-notify method for applications that are interested in things whose occurrences are hypothetic or unknown.

Functional decomposition. From early experiences, we know that 'context' ranges from simple raw data to highly structured symbolic information. This means that context must be elaborated and exploited at multiple levels of abstraction and complexity. A mature domain is characterized by the availability of canonical architectures that help structuring software systems. For example, in HCI, the Arch model makes explicit the lev-

els of abstraction of interactive systems, from the low-level physical interaction to the high-level domain-dependent functional core. In context-aware computing, no agreement has been found on a canonical decomposition. However, we progressively observe a distinction between context sensing at the lowest level of abstraction, followed by context transformation by interpretation and inference, and the expression of requests for services using high-level language: liquid in [15], and first order logic in [7].

Orthogonal to these functions, service discovery and recovery, history management, security, trust and privacy, need to be addressed. In CTK for example, the programmer must explicitly compose the federation of processes through the subscribe mechanism. When infrastructures go from local to global, they do so by interconnecting service providers. Clearly, this straightforward solution, prone to service failure, need to be improved with additional mechanisms such as redundancy used in IrisNet.

As for privacy, current solutions are still simplistic. The work on Confab provides an interesting approach, but at a high level of abstraction only. IrisNet, on the other hand, supports privacy at the sensing level only by filtering raw data.

2.2 Synthesis: Multi-scale Interoperable Run Time Infrastructures

Like Hong et al., we believe that the effective deployment of context-aware systems relies on the existence of run time infrastructures [4]. A run time infrastructure is a middleware that runs reliably and permanently to provide applications with 'public utility' services that would otherwise be developed for each application. In the light of our analysis, an infrastructure for context-aware computing must provide context services that are accessible and retrieved from anywhere on the planet, from the fix network infrastructure to spontaneous 'connectivity islands'. Context services are not floating items in a cloud, but form a fabric structured into multiple levels of abstraction.

Within a level of abstraction, the infrastructure must:
1. include mechanisms for on the fly creation of new services that do not disturb the existing fabric, that are inspectable and controllable through meta-interfaces and meta-data;
2. provide service authors with a set of core abstractions that comply with a single computational model and architecture style.

These general conceptual principles must be tuned and reconciled with practical considerations such as portability (e.g., operating system and language independence) and computation cost (e.g., memory footprint, latency, bit-rate communication, and power consumption). In ubiquitous computing, nodes range from small computers such as motes [3] to high-end computers. Their topology and operating systems may vary widely from TinyOS to conventional OS. Heavy-duty components are typically placed on high-end nodes. With 'connectivity islands', high-end computers may not be available. For small devices, a costly infrastructure is impractical. Therefore, one size infrastructure for context-aware computing does not fit all. Instead, we believe in a variety of interoperable infrastructures where each infrastructure is designed to fit the scale and functional coverage (i.e., level of abstraction) it is intended for.

3 The Contextor Infrastructure: Conceptual Aspects

The Contextor Infrastructure is a software concretization of the ontology presented in [11]. This model is two-fold: the levels of abstraction that the infrastructure covers, and the computational model used for each of the levels.

3.1 Levels of Abstraction

At the lowest level, the system's view of the world is provided by a collection of physical sensors. These sensors generate numeric values for *observables*. This corresponds to the *Sensing layer* (Cf. Figure 1). This layer, which is dependent on the sensing technology, encapsulates the diversity of sensors.

Observables may be numeric or symbolic. In order to determine meaning from numeric observables, the system must perform some series of transformations. This is the role of the *Transformation layer*. This layer is independent of the sensing technology and provides observables at the "right" level of abstraction.

The *Situation&Context Identification layer* is able to identify the current *situation* and *context* from observables and detect conditions for moving between situations and between contexts. It is the reasoning and inference layer.

The *Exploitation layer* acts as an adaptor between the application and the infrastructure. This is where applications express their requests for context services at a high level of abstraction. For performance reasons, the exploitation layer may skip the Situation&Context Identification layer and exchange information directly with the Transformation and Sensing layers. As for interactive systems, the slinky meta-model applies: some layers may not exist, and functions may shift between layers.

At every level of abstraction, one finds mechanisms and facilities to support privacy, trust, and security as well as history management, and discovery/recovery.

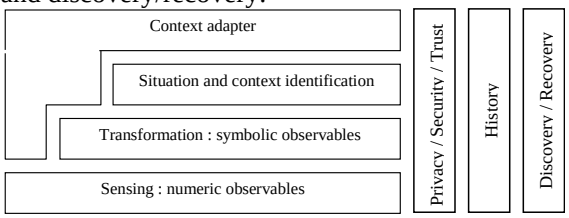

Fig. 1. Levels of abstraction for a complete general purpose infrastructure for context-aware computing.

According to our prescriptions from Section 2, layers define the scope for computational models and architectural styles. We have devised the Contextor computational model to address the Sensing and Transformation layers. This model is process-oriented. From the state of the art as well as from our own experience with the development of augmented environments and plastic user interfaces [12], we believe that the Situation&Context Identification and Exploitation layers are data-centric. In the following, we concentrate the description on the contextor model for which we have more definitive results.

3.2 The Contextor Computational Model

First, we introduce the notion of contextor, then, we present how contextors are assembled to cover the sensing and transformation layers of the infrastructure.

3.2.1 The Contextor

A *contextor* is a software abstraction that models a relation between observables. From the values of a set of observables, a contextor returns values for an observable. A contextor is comprised of a *functional core* and of a set of *typed communication channels* to receive and send values of observables.

Communication channels include the *data* and the *control* channels.

- Data channels are used to transfer values of observables between contextors: they are the interface of the contextor. Every value (whether it be a data-in or a data-out value) is associated with a meta-data. Meta-data is used to express the quality of the values exchanged between contextors. As mentioned in Section 2, in a world of uncertainty, the system must be able to auto-evaluate the quality of the services it provides. Therefore, every contextor evaluates its own behavior and exports this activity as meta-data.
- Control channels carry commands for inspecting and controlling contextors behavior: they are the meta-interface of the contextor. Commands cover both the functional and non-functional aspects of a contextor. For example, a contextor may receive a "Stop" command on its control-in channel because it has been recognized to be faulty by another one. Or, it may receive a QoS request that expresses the precision of the values expected on its data-out channel. Alternatively, based on the meta-data associated with the data received on its data-in channel, a contextor may decide to send a "Stop" command via its control-out channel.

The *functional core* of a contextor implements the function of the contextor using the values received on its data-in and control-in channels, delivers the result (decorated with meta-data) on its data-out channel, and may send some commands on its control-out channel [9].

Contextors with both input and output data channels are called *non-elementary contextors*. A contextor with no data-out channel is a *contextor adapter*. It is used as a bridge between an application (the Exploitation layer) and the Contextor Infrastructure. A contextor with no data-in channel is called an *elementary contextor*. It is used to encapsulate a physical sensor such as a thermometer, or any computational component that serves as source of physical data. For example, a video camera is encapsulated within an elementary contextor. Data-out corresponds to the video flow of images, its control-in allows other contextors to set the video rate, as well as the pan-tilt-zoom factors of the camera. Meta-data associated to data-out values expresses the resolution, the number of bits per pixel, resolution, and more generally, any factor that qualifies the images delivered by the contextor. Control-out is used by the contextor to acknowledge the execution of the commands received on the control-in channel.

Having defined the computational model of a contextor, we need now to specify how contextors are assembled.

3.2.2 Federations of Contextors

In good software engineering practice, the assembly of software components is ruled by an architecture style. In our case, the vocabulary is defined by the set {contextor, data-in, data-out, control-in, control-out}. What we need to express through the composition of these vocabulary elements is sequences of transformations on observables, from numeric data to symbolic information. This functional requirement maps easily onto the *data-flow* model.

Contextors are composed by connecting data-in channels with compliant data-out channels. Two channels are compliant if they convey data of the same type. A *source contextor* is a contextor that provides another contextor with data-in values. A *sink contextor* is a contextor that receives data-out values from another contextor. In order to support sharing, the data-out channel can simultaneously feed into multiple sink contextors. The control-in channel of a contextor is connected to the control-out channel of its sink contextors. Consequently, the control-out channel of a contextor is connected to the control-in channel of its source contextors.

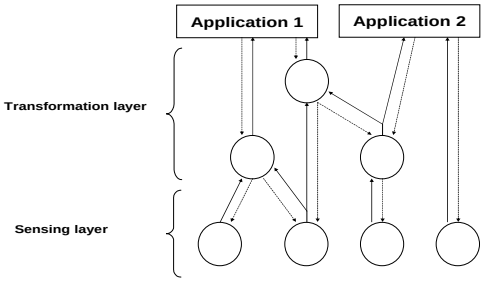

Fig. 2. Two federations of contextors assembled according to the data-flow architecture style.

Figure 2 shows examples of federations of contextors. Source interactors at the base of the federation are elementary contextors. They form the sensing layer of the infrastructure. The other contextors are part of the transformation layer of the infrastructure. Sink contextors at the top of the federation provide symbolic observables to applications via context adaptors or to any upper level of the infrastructure.

Connections between source and sink contextors may be static (i.e., wired by implementation), or semi-static (i.e., computed at run time when the system is launched), or transient (i.e., may change dynamically). Static connections correspond to well-packaged immutable compositions that work together in a tightly and optimal fashion. Transient connections allow for the dynamic discovery/recovery of contextors. They support the discovery/recovery functions for the sensing and transformation levels of the infrastructure.

In the next section, we show how the conceptual computational model has been turned into an operational infrastructure for the sensing and interpretation layers.

4 Implementation of the Sensing and Interpretation Layers

Contextors are implemented as Java P2P components, using the UDP multicast communication protocol for service discovery and TCP direct links once source and sink contextors have successfully negotiated their cooperation. Messages are XML documents that ensure interoperability and extensibility. Contextors are distributed over a network whose horizon can range from a single machine to the planet. Figure 3 illustrates the life cycle of a contextor. In Section 4.1, we present the role of the components of a contextor as we go through its life cycle. Then in 4.2, we discuss the geographical scope of the infrastructure.

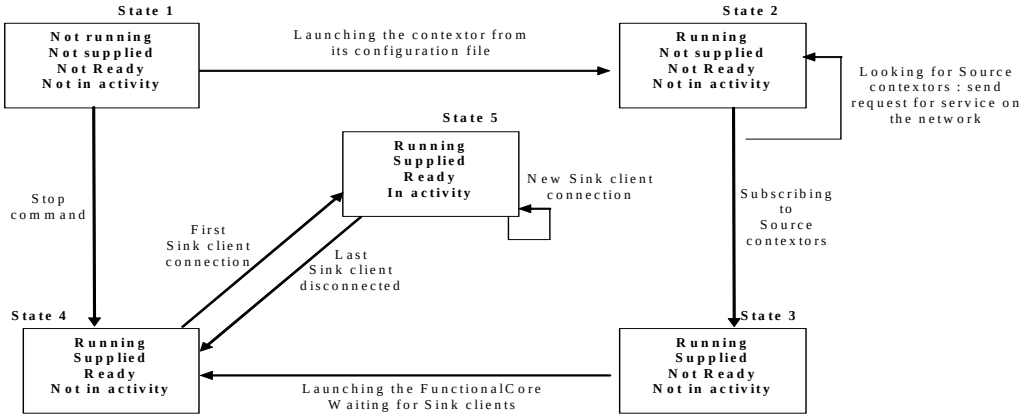

Fig. 3. Life cycle of a contextor.

4.1 Life Cycle of a Contextor

At the beginning (State 1), the contextor resides on a node of the network (i.e., a machine) as an XML document called a "configuration file". This file specifies, among other things, the class name of the contextor, the geographical location of the node it currently resides on, the network it belongs to, the control and inspection commands it accepts, the data types (and meta-data) of its data-in channels as well as the desired mode for data acquisition, the data type (and meta-data) of its data-out channel as well as the maximum number of sink contextors it can serve simultaneously, a login and password to control the access right to the data it delivers, and the mode of delivery of information. Data may be delivered/acquired *once* (the contextor then returns to State 1); it may be delivered *on request only* to support the query-answer method, or *on change* to support the subscribe-notify approach; it may also be delivered *periodically* or each time it is *re-computed* by the contextor.

When the contextor is launched, its InternalContextorDescription component initializes the contextor from its "configuration file": if the contextor is not an adaptor, a DataOut object, its associated MetaData and DataExchangeMode objects are created. These three objects model the data-out channel. Similarly, if the contextor is not elementary, a DataIn, MetaData and DataExchangeMode are created for each one of the data-in channels. The contextor is now in State 2: it is running but its data-in channels are not connected its data-out channel has no sink client (the contextor is not supplied, not ready, and not in activity).

Our contextor (let's call it C), now looks for source contextors to feed its data-in channels. To do this, its DataInManager broadcasts a request on the network using a UDP socket. This request specifies the contextor identification, the data type of its data-in channels as well as the desired DataExchangeMode. Contextors of the network that can satisfy one (or multiple) of the data-in channels, create a direct link with C. These direct links are managed by the DataInManager of C that dispatches the answers received from its potential source contextors to the appropriate DataIn object. If it does not receive any satisfactory answer, a DataIn object periodically broadcasts a request that describes its own need. A DataIn that receives an acceptable answer from a contextor S, creates a SourceThread that manages the communication with S.

In State 3, *C* is running and fully supplied with input data. Therefore, its FunctionalCore can start computing input data to deliver results. *C* enters state 4: it is running, supplied, in activity, but has no sink contextor yet. It creates one DataOutThread per contextor it can potentially serve, then waits for the service requests that are broadcasted on the network. When *C* receives a service request, this request is transmitted to a free DataOutThread. In turn, this DataOutThread creates an InternalRequestDescription whose role is to check the request description with the services that *C* can provide. If *C* can provide one (or several) of the requested data, a direct link TCP socket is opened with the sink client as well as a ControlThread so that the sink client can inspect and control *C*. When the link is opened, the DataoutThread sends a message to the sink contextor. This message specifies the identification of *C*, the data it is able to support among the requested data, as well as the DataExchangeMode that it can support among those requested by the sink. Then, the DataOutThread waits for an "identification login-password" command to check that the sink contextor has the right to use the data-out channel. If the verification is successful, *C* enters State 5: it is running, supplied, ready, in activity. It returns to State 4 (running, supplied, ready, not in activity) when its last sink contextor disconnects. From there, it returns to State 1 if a "stop" command is received.

So far, we have described how the contextors that are installed and launched on a particular node build federations from contextors that belong to the same network. We now discuss the geographical scope (or horizon/range) of this network.

4.2 Geographical Scope of the Infrastructure

In our current implementation, the scope of the infrastructure is defined by tuning the TTL parameter of the UDP multicast protocol:

- TTL=0 corresponds to the machine where the contextors reside. This setting is used for embedded context-aware systems such as the "Squeeze Me, Tilt Me" tangible user interface where a PDA augmented with orientation and tactile sensors can be tilted and squeezed to control document scrolling.
- TTL=1 denotes a network whose horizon is limited to proximal areas (Local Area Network). Clearly, this approach provides an approximate solution to the problem of ad hoc networks whose boundaries need sometimes to be better specified. Although important, we have left this problem aside, and will reuse solutions like the "proximity group" techniques as they will become available [14].
- TTL=2 or higher, corresponds to a large horizon. This is where an application is interested in source data that is far from the current location. Although this option provides a simple mechanism, it may lead to network congestion. To avoid flooding, we propose an hybrid P2P hierarchical architecture illustrated in Figure 4. This solution is under implementation.

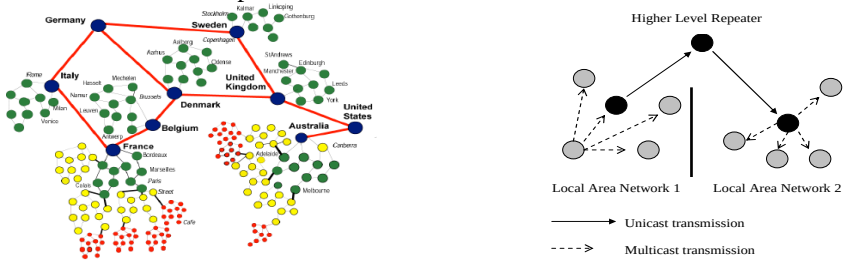

Fig. 4. An Hybrid P2P hierarchical architecture inspired from [13]. a) An example of network topology at the planet level. b) A mix of multicast transmissions at the local level, and unicast transmissions at the global level. Grey circles denote contextors, whereas black circles represent repeaters.

As shown in Figure 4a), the planet is populated with repeaters that match the geographical structure of the world in terms of Continent/Country/State/Province/City/District/… At the lowest level, a local area network repeater is used as a gateway between the contextors of the area it serves and the planet. As shown in Figure 4b) the local area contextors use the UDP multicast TTL=1. Due to the nature of the protocol, they do not need to have an explicit knowledge of the repeater. Repeaters use unicast transmission for long distances. As specified in the previous section, the location attribute of a contextor (of the form Continent/Country/State/Province/City/District/…), allows repeaters to perform the appropriate routing.

5 Conclusion

In summary, we have proposed a set of criteria for comparing current infrastructures for context-aware computing, we have proposed a four-layer functional decomposition as a tentative reference model, and we have presented the Contextor Infrastructure (CI) as a technical solution for the first two levels of abstraction: the sensing and transformation layers. CI is the result of a top-town design approach informed by the theoretical foundations of a simple ontology [11], and an abstract computational process model.

Unlike previous work, in CI, 1) The horizon for contextors discovery is controllable, from the embedded context-aware system to the planet level; 2) For local horizons, the full P2P discovery of contextors permits to support 'connectivity islands'; 3) privacy is provided in a fine-grained manner at the contextor level via a login-password mechanism; 4) Data acquisition includes a rich set of parameters (once, onChange, onRequest, periodically, etc.), 5) The automatic construction of contextors federations supports dynamic reconfiguration in case of failure. CI has been used successfully in the development of I-AM. Next steps include: performance evaluation and development of situation&context identification layer applied to the adaptation of UIs to platforms, places, and people.

References

1. Winograd, T.: Architectures for Context, Human-Computer Interaction, 16:2-3, (2001).
2. Dey A.K., Salber D., Abowd G.D., A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, anchor article of a special issue on Context-Aware Computing, in the Human-Computer Interaction Journal, Vol. 16, (2001).
3. Gibbons P.B., Karp B., Ke Y., Nath S., Seshan S.. IrisNet: An Architecture for a Worldwide Sensor Web. IEEE Pervasive Computing Mobile and Ubiquitous Computing, Special Issue Sensor & Actuatirs Networks, Vol 2(4), pp.22-33, (Oct-Dec. 2003).
4. Jason I. Hong and James A. Landay, An Infrastructure Approach to Context-Aware Computing. In Human-Computer Interaction ,Vol. 16, (2001).
5. Glassey R., Stevenson G., Richmond M., Wang F., Nixon P., Terzis S., and Ferguson I.: Towards a middleware for generalised context management. in 1st MPAC03, (June 2003).
6. Judd, G, Steenkiste: Providing Contextual Information to Pervasive Computing Applications P.IEEE International Conference on Pervasive Computing, March 23-25, (2003).
7. Christopher K. Hess and Roy H. Campbell, A Context-Aware Data Management System for Ubiquitous Computing Applications in International Conference of Distributed Computing Systems (ICDCS 2003), Providence, Rhode Island, (May 19-22, 2003).
8. Kiczales G., Towards a New Model of Abstraction in Software Engineering, Proc. IMSA'92 Worshop on Reflection and Metalevel Architectures, Tokyo, (1992).
9. Coutaz J. and Rey G., Foundations for a Theory of Contextors, in Computer Aided Design of User Interfaces, Springer Verlag, (June 2002).
10. Hinckley K.: Synchronous gestures for multiple persons and computers. UIST03: 149-158
11. Crowley, J. L., Coutaz, J., Rey, G., Reignier, P., "Perceptual Components for Context Aware Computing", UBICOMP 2002, Goteborg, Sweden, (September 2002).
12. Coutaz J., Lachenal C., Rey, G, Barralon N.: Reference Framework for Multi-surface Interaction, Deliverables D17 to D20, Project GLOSS IST-2000-26070, (2002 - 2003).
13. Dearle, A., et al.: Architectural Support for Global Smart Spaces. In Proc. 4th International Conference on Mobile Data Management, Melbourne, Australia, pp 153-164, (2003).
14. Singh K., Clarke S., Nedos A., Cahill V.: Proximity Groups for Mobile Ad Hoc Networks. In proc. of the "Pervasive Computing" workshop at OOPSLA (2002).
15. Heer J., Newberger A., Beckmann C. and Hong J.: liquid: Context-Aware Distributed Queries , UBICOMP pp (2003).
16. Johanson B., Fox B., Winograd T.: The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing Magazine, (April-June 2002).