A Programmable Context Interface to Build a Context Infrastructure for Worldwide Smart Applications

Kyung-Lang Park, Chang-Soon Kim, Chang-Duk Kang, and Shin-Dug Kim

Super Computing Lab, Department of Computer Science, Yonsei University 120-749,
134 Shinchon-Dong, Seodaemoon-Gu, Seoul, Korea
{lanx, flsoon, niceguy, sdkim}@yonsei.ac.kr
http://supercom.yonsei.ac.kr

Abstract. Context-awareness is one of the most important technologies for the ubiquitous computing. To embed such a technology into applications, first of all, a well-defined architectural framework is required to support applications to obtain necessary contexts. Several studies have been proposed, but most of them overlooked one fundamental feature that contexts are subjective things and context-awareness is subjective behavior. Thus, providing pre-defined, pre-programmed operators which generate contexts by combining a couple of sensed data is very impractical. Instead, in this paper, we provide a well-defined programmable interface to applications, so that they can obtain and use contexts according to their own ideas. To evaluate our architecture and context interface, we implement all components comprising of the context infrastructure and perform several experiments. The results show the proposed method provides a flexible and expressive interface, but does not countervail the performance.

1 Introduction

Ubiquitous computing [11] is being considered as a next generation computing paradigm these days. Computer-based technologies will be embedded into our casual life ubiquitously, so that we can use a variety of smart services anytime and anywhere. One of the most important technologies of the ubiquitous computing is context-awareness. If a service perceives our situational contexts, it will operate much smarter without any artificial intelligence stuff.

The most fundamental idea is to insert various sensors into an application. Then, the application can be aware of our situation by analyzing sensed data. However, such a sensor-driven approach gives developers pain because the developers have to know about all the sensors in detail. They have to spend more time on sensors than the core logics of applications. Several approaches have been proposed to overcome the problem. They attempted to separate context handling modules from applications and provided convenient toolkits and frameworks to develop sensor-driven smart applications much easier [1, 2, 7, 15, 16]. However, they are not sufficient enough to support the development of enormous context-aware applications. Nowadays, thus, an approach to establish a context infrastructure for innumerable context-aware applications is on the rise. When a context infrastructure is constructed, we can share a lot of sensors, information sources, and processing components. Consequently, we will design smart applications without any difficulty [8].

Some studies are based on the infrastructure approach [4, 12, 13]. They assume a worldwide model and consider various and enormous context-aware applications which use shared sensors in the world. Although they addressed some infrastructure issues such as scaling up, information dissemination, and unified sensor abstraction, they overlooked one important feature that contexts are subjective things rather than objective. To generate a context, several sensed data have to be aggregated. However, the problem is that the developers of the context infrastructure and developers of applications are different. Even though a well-defined component in the context infrastructure aggregates sensed data and generates contexts with a novel algorithm, they cannot be accepted to some applications. Thus, it is required to support applications by specifying their ideas to generate and use contexts.

In this paper, we propose an architecture for the context infrastructure and describe how applications obtain and use contexts subjectively. Our context interface provides a context description language (CDL) for applications to specify their requests. Thus, based on the CDL given by the application, the infrastructure discovers proper sensors, aggregates sensed data, generates contexts, and sends them to the applications. To evaluate our architecture and context interface, we implemented all components comprising of the context infrastructure and did several experiments. Experimental results show that the proposed method provides a flexible and expressive interface, but does not countervail the performance. Rest of the paper is organized as follows: In Section 2, we introduce several related work. Section 3 describes our context infrastructure and Section 4 explains the programmable context interface which is the core of our infrastructure. Section 5 provides several experimental results to evaluate our research. Finally, we conclude in Section 6.

2 Related work

There are several studies to support development of context-aware applications. The Context Toolkit [1] is a pioneering work of this area. It suggested a well-defined conceptual framework that supports context-aware applications and provided several software components which aggregate and manage sensed data. Thus, developers can make context-aware applications rapidly by using the Context Toolkit. Since the advent of the Context Toolkit, a number of researches have been appearing. [2, 5, 7, 9, 15, 16].

Those are very convenient to develop a few number of applications in a domain, but not sufficient to support a large number of applications in multiple domains. Hong's article pointed out such a problem and introduced an infrastructure approach to develop a large number of context-aware applications [8]. There are several projects based on the infrastructure approach. Chen et al. proposed Solar, an infrastructure for context-aware applications [4]. Solar generates contexts by using an operator graph, where an operator is a self-contained data-processing component. A higher-level operator can be made by composing existing operators. Because Solar considered a large-scale context fusion network, it addressed several infrastructure issues, scalability and addressing. Contextor Infrastructure [6] took up the similar fashion. It suggested contextors as a building block of the infrastructure, where a contextor is a component which can provide contextual information. Either a sensor or a processing unit like an operator of Solar can be a contextor. They suggested applications retrieved contexts by composing several contextors. SCINet (Strathclyde Context Infrastructure) [13] and NEXUS [12] designed local context servers and extended it to be used for global areas by composing network of servers. SCINet comprises of CSs (Context Server) which maintains a central store of contexts and provides the access point for the applications. When a CS receives a query from an application, a query resolver in the CS generates a path to make the required context. In NEXUS, they defined an AHSS (Aware Home Spatial Service), a kind of context server, which is suitable for mid-size home environments and bridged AHSSs by using a query interface named AWQL. Semantic Space [17] and Gaia's context infrastructure [3] are also aimed to construct an infrastructure for context-aware applications. Semantic Space used context wrappers as a building block of the infrastructure. It also leveraged technologies from the Semantic Web. It provides RDF-based query language to combine sensors and existing semantic web information. Gaia project also assumed that there would be a lot of context providers and defined how context providers generate contexts.

Our work has the same perspective with above researches which are based on the infrastructure approach, but we do not agree with them in terms of context fusion and context inference in the infrastructure. We do not allow any kind of data composition predefined in the infrastructure. Instead, we provide more flexible and programmable interface to applications, so that they can obtain and use subjective contexts.

3 Architecture for a Context Infrastructure

Most of all, a well-defined architecture is required to build a context infrastructure for worldwide smart applications. In infrastructure approach, sensors are not for a specific application. Thus, we should define a bottleneck protocol to glue sensors onto an application. Figure 1 shows our architecture for the context infrastructure. We divide the bottleneck protocol into two layers. One is *abstraction layer* and the other is *collective layer*. The abstraction layer hides complexity of the actual sensors by wrapping them with a single software interface and publishes sensors to the outside. Typically, sensors use different I/O interfaces and express sensed data by using different data models and formats. Thus, it is quite difficult for an application to understand several kinds of I/O interfaces, data models, and formats. When sensors are abstracted, upper-layer components can access to the sensors through a single interface without deep knowledge of each sensor.

Our virtual sensor is based on Web services. It is based on fully open and standard protocols and does not depend on the physical communication method. A virtual sensor expresses sensed data as a tuple which comprises of *entity*, *location*, *time*, *attribute*, and *value*. The value is actual sensed data and others are kind of meta-information to help applications to understand the value. Upper-leyer components have only to know the Web service interface and the tuple model of the virtual sensor to access various sensors.

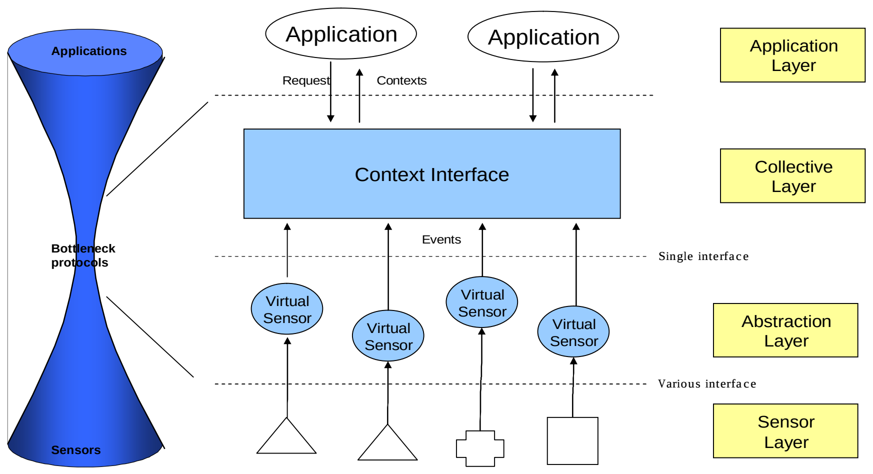
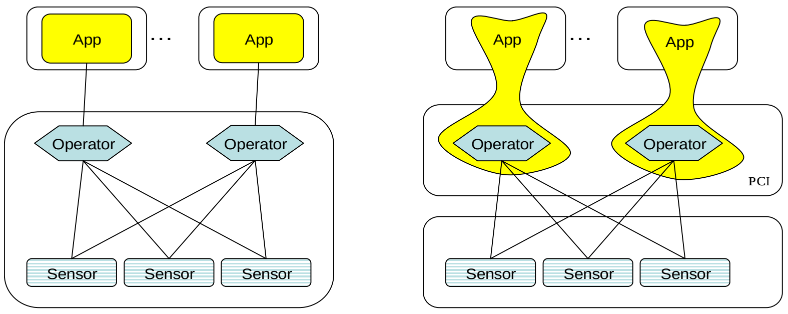


Fig. 1. The proposed architecture for the context infrastructure.

However, abstraction layer is not sufficient to support context-aware applications. First of all, applications do not know where proper sensors are. Thus, discovery service has to be provided to applications. Also, most of sensors are immature. They provide imperfect information and lack the confidence of information. Thus, they have to be combined to provide necessary information to applications. Both functionalities, i.e., sensor fusion and sensor discovery, are provided by the collective layer. It receives a request for contexts from an application, aggregates information from published virtual sensors, generates contexts by analyzing the information, and sends the contexts to the application. At this point, we should remind that contexts are subjective representation of information and context-awareness is a subjective behavior. If the collective layer generates contexts with its own algorithm, the contexts are not subjective products of the application but of the collective layer. Therefore, contexts can be used to only a few applications which agree with them. When we make a single application, it does not matter, but it cannot be used for the context infrastructure. Thus, it is more efficient to provide a method for applications to specify how to aggregate and generate necessary contexts. It is the programmable context interface (PCI) proposed in this paper. Figure 2 shows the difference between two. When the collective layer provides predefined operators, they are not a part of an application. Applications have to decide whether they use the operators or not. When using the PCI, operators can be a part of applications. Thus, applications can generate their own contexts subjectively.



(a) Predefined operators (b) the proposed PCI model

Fig. 2. Predefined operators vs. the PCI model.

4 Programmable Context Interface

As pointed out in the previous sections, the core of the context infrastructure is the programmable context interface. By using it, applications can generate and utilize contexts subjectively. In this section, we describe the PCI and its components in detail.

It consists of three major components: application context managers (ACMs), the context broker (CB), and the channel manager (CM). The ACM is responsible for managing contexts for an application. At first, an application should generate a CDL to specify its request and submit to the context interface (1). The ACM factory receives the CDL and creates an ACM for the application (2). The ACM parses the CDL and constructs a tree structure based on the CDL (3). The CDL tree collects events from sensors, generates contexts, and sends them to the application. In the CDL, a sensor is specified by using either an end-point address of the sensor or a query to discover the sensor. If a query is used to describe a sensor, it will be sent to the CB after the ACM constructs the tree (4). Then, the CB discovers a proper sensor and sends the end-point address of the sensor to the ACM (5). When this stage is over, all sensor descriptions are filled with the specific addresses of sensors. It is called the grounded CDL. After that, the ACM sends the grounded CDL to the CM (6). The CM subscribes virtual sensors instead of the ACM (7). When an event from a virtual sensor arrives at the CM (8), it sends the event to the ACM (9). The CDL tree in the ACM receives the event and operates it according to the operation defined in the CDL (10). This operation flow is shown in Figure 3.

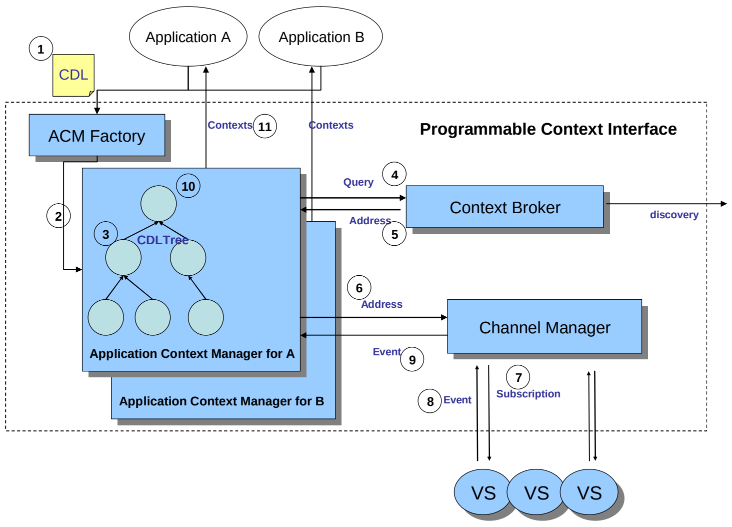


Fig. 3. Operation flow of the Programmable Context Interface.

The specification of the CDL is the most important part of the PCI. By using the CDL, application can specify a lot of things to generate contexts by using the CDL. They can specify which sensors they use, how aggregate sensed data, and how generate the context from the sensed data. Thus, it has to be simple, but expressive enough to specify any operation required for applications. Figure 4 shows an example of the CDL.

```
<?xml version="1.0" encoding="UTF-8"?>
<cdl:Definitions xmlns:cdl="http://supercom.yonsei.ac.kr/2005/03/cdl090" name="Park Presence">
 <cdl:SensorList>
  <cdl:Sensor type="constant" name="S_ONE">
    <cdl:Query num="1">
                <cdl:Tuple>
                    <cdl:Location/>Z1</cdl:Location>
                    <cdl:attribute ontologyRef="http://supercom.../2005/03/ont/test/Pressure">Pressure</cdl:Attribute>
                <cdl:Tuple>
    </cdl:Query>
    <cdl:Parents num="1">
      <cdl:Parent name="OP_ONE"/>
    </cdl:Parents>
  </cdl:Sensor>
  <cdl:Sensor type="subscription" name="S_TWO">
    <cdl:Address>
       <cdl:URI>http://121.141.10.199/WS/Sensors/ID_Sensor.wsdl</URI>
       <cdl:Parameter/>
     </cdl:Address>
    <cdl:Parents num="1">
      <cdl:Parent name="OP_ONE"/>
    </cdl:Parents>
  </cdl:Sensor>
 </cdl:SensorList>
 <cdl:OperatorList>
   <cdl:Operator name="OP_ONE">
    <cdl:Inputs num="3">
                <cdl:Input name="S_ONE" driver="true"/>
                <cdl:Input name="S_TWO"/>
    </cdl:Inputs>
    <cdl:Operation type="mapping">
                <cdl:Mapping>
                               <cdl:Condition>
                               <cdl:Cmd>$S_ONE.value > 1 </cdl:Cmd>
                               <cdl:Cmd>$S_TWO.value == Park</cdl:Cmd>
                               <cdl:Tuple>
                                              <cdl:Entity>Park</cdl:Entity>
                                              <cdl:Location>Z1</cdl:Location>
                                              <cdl:Time type="new">
                                              <cdl:Attribute>Presence</cdl:Attribute>
                                              <cdl:Value>true</cdl:Value>
                               </cdl:Tuple>
                               </cdl:Condition>
                               <cdl:Condition>
                               <cdl:Tuple>
                                              <cdl:Entity>Park</cdl:Entity>
                                              <cdl:Location>Z1</cdl:Location>
                                              <cdl:Time type="new">
                                              <cdl:Attribute>Presence</cdl:Attribute>
                                              <cdl:Value>false</cdl:Value>
                               </cdl:Tuple>
                               </cdl:Condition>
                </cdl:Mapping>
     </cdl:Operation>
    </cdl:Operator>
   </cdl:OperatorList>
   <cdl:RequestList>
                <cdl:Request type="operator" name="OP_ONE"/>
   </cdl:RequestList>
</cdl:Definitions>/
```

Fig. 4. An example of the CDL. It represents a kind of person presence sensor

A CDL comprises of three parts, a list of sensors, a list of operators, and a list of requests. <cdl:sensorList> specifies sensors which are going to be used. Sensors can be categorized into two types, i.e., subscription and constant. The constant means that the sensor retrieves sensed data only once, but the subscription means that the sensor retrieves data continuously. A sensor can be specified by either <cdl:query> or <cdl:address>. When the application already knows the sensor, <cdl:address> is used. If the application does not know about the sensor, <cdl:query> has to be used to discover a group of proper sensors. The query is based on the context tuple we defined. The query element is sent to the context broker (CB) and the CB transforms the query element into an address element. In this example, we specified two sensors. One is specified by a query element. The other is specified by an address element.

<cdl:operatorList> specifies operators. An operator is a kind of processing module. It receives data from sensors or other operators and processes the data with own operation. It can be defined by <cdl:inputs> and <cdl:operation>. The <cdl:inputs> specifies sensors and operators that have to send data to the operator. Only input elements marked as the driver can trigger the operation. The operation in an operator is specified by the <cdl:operation>. Current version of the CDL supports five types of operations. Those are mapping, selection, integration, calculation, and manual. Finally, <cdl:Request> is used to mark sensors or operations whose output would be sent to the application. When all nodes marked as request finish their operation, the ACM makes a result by combining events generated from requested nodes and sends it to the application.

4 Evaluation

In this section, we evaluate the proposed context infrastructure based on the programmable context interface. The main difference between previous approaches and our approach is that we do not use pre-defined operations. Instead, we provide an interface to applications to program operations that they need to obtain necessary contexts. Thus, we have to show that our operations can be performed in the same way as the pre-defined and pre-programmed operations.

In our approach, an ACM should construct a CDL tree on the fly. If there is heavy overhead to construct a tree, it cannot be used even though it provides a very flexible and convenient interface. This is because contexts are time-sensitive information. Following experimental results show the construction of a CDL tree does not take much time although there are a number of nodes specified in a CDL. They also show that overall response time of the PCI is as fast as other systems which use predefined, pre-programmed model.

4.1 Experimental results

All components comprising of the context infrastructure are implemented by using Java SDK 1.5 and WSDP (Web Services Development Package) on Window XP machines. ACMs, the CB, the CM are located in the same machine, and each sensor and application are placed in different machines. A machine with the PCI components is Pentium IV 1.33GHz with 1GB main memory. All machines are connected by local area networks. We firstly make a CDL which consists of two sensors and one operator, and measure the time at several points to see the basic timing diagram. Figure 5 shows the result of the experiment. It takes about 400ms from an application submitted a CDL to the application received the result. Specifically, such a time delay is quite feasible because valid time period of a sensed data is generally a couple of seconds. The time from application launched to the time the ACM received the CDL stands for the communication time between two machines. It is not related to the PCI and depends on the network status. From the time to the CDL tree constructed stands for the tree construction time. It is very important to evaluate the ACM. Thus, we will show other experimental results about the construction time. The next one is the time to discover sensors. It depends on the ability of the CB. The next is the time to subscribe to the sensors. It is very simple operation and depends on the network status rather than the ability of the CM. The time from all events arrived to the result stands for the processing time. It is also very important. The operator generated in the ACM interprets the operation specified in the CDL on the fly. Thus, it takes much more time compared to the pre-programmed operations. Finally, the time for the application to receive the result is also a kind of communication time between the PCI host and the application.

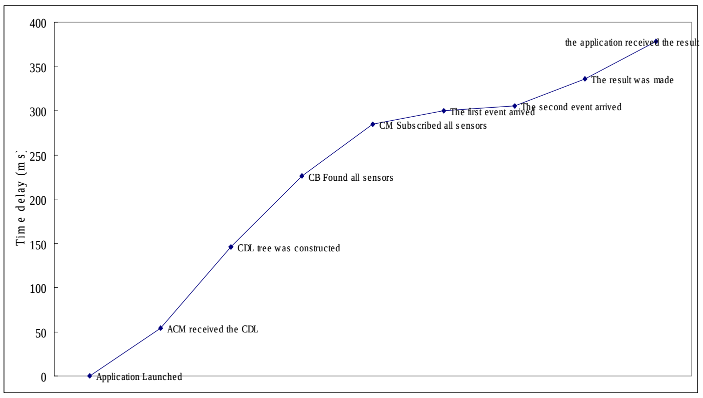
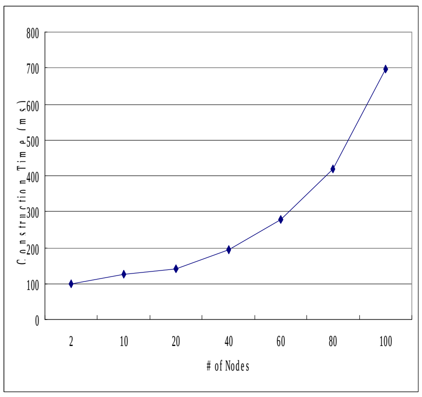


Fig. 5. The timing diagram when an application submits a CDL consisting of two sensors and one operation.

The most important point in this time diagram is the construction time, because our approach generates operation trees in running time, which may not exist in other approaches. To measure the construction time, we perform additional experiments. We measure only construction time with changing the number of nodes. Figure 6 shows the result of these experiments. When the number of nodes is two, it takes only 97ms. It takes more time as the number of nodes increases. When the number of nodes becomes more than one hundred, the construction time will be less than one second. However, one CDL include about 10 to 20 sensors typically. It takes only 120ms, so that it does not cause any heavy overhead to the application



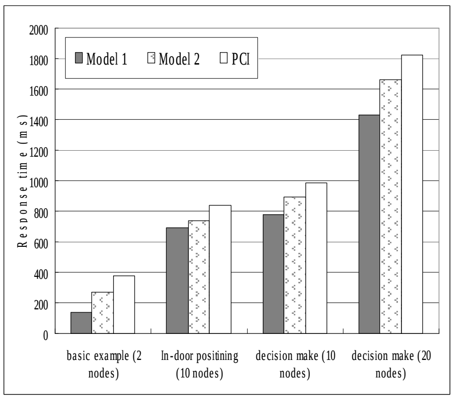


Fig. 6. CDL tree construction time as changing the number of nodes (left). Response time compared to other models (right)

Finally, four context-aware applications are designed by using several context acquisition models. The first model is pre-programmed and pre-defined model. In this model, operators are already developed and published, and applications already know about the operators. Probably, the programmer of operators and the programmer of applications should be same. The Context Toolkit uses this approach. The next one is pre-programmed and not-defined model. Several useful operators are provided by some participants, but the application should discover the operators. A lot of approaches based on the infrastructure such as Solar and GAIA use this model. Our PCI model is not-programmed and not-defined model. There is no pre-programmed operator. Instead, applications have to generate subjective operators by submitting CDLs. We compare the response time of three models. Definitely, the first model shows the best performance as shown in Figure 6. It is because there is no additional overhead in the model such as construction time and discovery time. The second model only has discovery time as overhead. However, there is no huge difference between three models in terms of the performance. Simply, it means that the construction time and discovery time occupy a small portion of the response time. But, the PCI model provides more convenient and flexible interfaces than other models. Thus, it is more feasible approach to support a number of context-aware applications.

5 Conclusion

In this paper, we have presented an architecture for supporting context-aware applications based on the infrastructure approach. The contribution of this paper is that we suggest a method that applications obtain and use contexts subjectively. When an infrastructure provides predefined and pre-programmed operators to extract contexts by using sensed data, it cannot be used for many applications. It is more efficient to provide a programmable interface to applications to specify their requests for contexts.

Experimental results show that it provides feasible response time compared to other context acquisition models.

References

1. A. K. Dey, D. Salber, and G. D. Abowd,: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, Human Computing Interaction. Vol. 16. (2001) 97-166
2. A. Schemidt, U. Karlsruhe, K. V. Laerhoven,: How to build Smart Applications, IEEE Personal Communications. (2001)
3. A. Ranganathan and R. H. Campbell.: An infrastructure for Context-awareness based on first order logic, Personal and Ubiquitous Computing, Vol. 7. Springer-Verglag London (2003) 353-364
4. G. Chen, M. Li, and D. Kotz.: Design and Implementation of a Large-Scale Context Fusion Network. In Proc. IEEE Internation Conference on Mobile and Ubiquitous Systems. (2004)
5. G. Judd and P. Steenkiste.: Providing Contextual Information to Ubiquitous Computing Applications. In Proc. IEEE Conference on Pervasive Computing and Communiations. (2003)
6. G. Rey and J. Coutaz.: The Contextor Infrasturcture for Context-aware Computing. In Proc. Workshop on the Component-oriented approach to context-aware systems (ECOOP04). (2004)
7. H. Lei, D. Sow, J. Davis, G. Banavar, and M. Ebling.: The Design and applications of a Context Service. Mobile Computing and Communications Review, Vol 6, Num. 4.
8. J. Hong and J.A. Landay.: An Infrastructure Approach to Context-Aware Computing, Human Computer Interaction, Vol. 16. (2001) 287-303
9. K. Koh, C. Choi, K. Park, S. Lim and S. Kim.: A Multilayered Context Engine for the smartHome. In Proc. International Conference on Computer Science, Software Engineering, Information Technology, E-business, and Applications. (2004)
10. M. Benerecetti, P. Pouquet, M. Bonifacio.: Distributed Context-Aware Systems. Human Computer Interaction. Vol. 16 (2001) 218-228.
11. M. Weiger.: The computer for the 21st Century. Scientific American, Vol. 265, No. 3. (1991) 66-75
12. O. Lehmann, M. Bauer, C. Becker, and D. Nicklas.: From Home to World – Supporting Context-aware Applications through World Models. In Proc. IEEE Conference on Pervasive Computing and Communications. (2004)
13. R. Glassey, G. Stevenson, M. Richmond, P. Nixon, S. Terzis, F. Wang, and I. Ferguson.: Towards a Middleware for Generalised Context Managerment. In Proc. MPAC (2003)
14. S. Jang and W. Woo.: Ubi-UCAM: A Unified Context-Aware Application Model. Lecture Notes in Artificial Intelligene, Vol. 2680. Spring-Verlag. (2003) 178-189.
15. T. Winograd.: Architectures For Context. Human Computing Interaction, Vol. 16. (2001) 401-419
16. W. N. Shilit.: A System Architecture for Context-Aware Mobile Computing. Ph.D. Thesis, Columbia University (1995)
17. X. Wang, J.S. Dong, C. Chin, S. Hettiarachchi, and D. Zhang.: Semantic Space: An infrastructure for Smart Spaces, Pervasive Computing IEEE, Vol 03, Issue 3. (2004)