Scalable Middleware for Context-Aware Ubiquitous Computing

YOUNG-CHUL SHIM, HO-SEOK KANG, SEHUN NO
Department of Computer Engineering
Hongik University
72-1 Sangsudong, Mapogu, Seoul
Republic of Korea

Abstract: - In ubiquitous computing systems it is important for applications to obtain context information and adapt their behavior according to context information. For a ubiquitous application to obtain and use context information, it is necessary to provide basic services to discover proper context providers and exchange/store/infer from context information. Moreover, when providing such services, non-functional issues such as scalability, security, heterogeneity, etc. should also be considered. In this paper we introduce the architecture for a middleware providing basic services. This middleware makes it easier for ubiquitous applications to obtain context information and for sensor networks to provide context information. We also explain how the proposed middleware addresses the non-functional issues.

Key-Words: - Context-Awareness, Ubiquitous Computing, Middleware, Distributed Computing

1 Introduction

In ubiquitous computing nearly everything is connected to the network in order to provide service and information on anything anywhere and anytime. For ubiquitous applications, it is important to be able to perceive context information on surrounding environments and adapt their behavior according to context information. Some of the examples for contexts include the location of a person, time, temperature, the velocity of an object, etc. An entity which obtains and uses context information is called a context consumer while an entity providing context information is called a context provider. Sensor networks are typical examples of context providers.

But context consumers and providers can encounter many problems while trying to obtain and provide contexts. Context consumers must discover proper context providers and exchange context information using some communication protocols. Moreover, context consumers and providers can move their location during operation. It is also important to process context information efficiently and securely in spite that there are a large number of context consumers and providers of many different hardware and software types.

* This research was supported by the MIC Korea under the ITRC support program supervised by the IITA and also supported by the second Brain Korea(BK) 21 Project in 2006

Early research works on the middleware for context-aware ubiquitous computing includes Context Toolkit[1] and Gaia[2]. They define core services and frameworks for building distributed context-aware applications. Their core services are discovering context providers, storing and exchanging context information, and inferring from simple context information but they do not address other essential issues such scalability and security. The Solar system implements context fusion networks and tries to enhance the scalability by enabling sharing of partial results obtained during the evaluation of context requests[3]. The Confab is a toolkit which facilitates the construction of privacy-sensitive ubiquitous applications[4]. While the Solar system focuses on only some aspects of scalability, the Confab toolkit focuses on only the privacy aspect of security. The Pace system tries to address most of the important issues in middleware but it is not quite clear whether their solution is adequate and sufficient especially in the area of scalability and security in spite of the developer's claim[5].

In this paper we propose a new middleware architecture for context-aware ubiquitous computing. The proposed middleware architecture provides mechanisms for not only discovering context providers and exchanging/storing/inferring from context information but also enhancing scalability through the distributed processing of context requests and guaranteeing the integrity and privacy of

exchanged context information. The rest of the paper is organized as follows. Section 2 lists the requirements of middlewares for context-aware ubiquitous computing systems and Section 3 describes the proposed middleware architecture. Section 4 explains how the new architecture supports the identified requirements and is followed by the conclusion in Section 5

2 Requirements of the Middleware

In this section we explain requirements of the middleware for the context-aware ubiquitous computing. Henricken et al summarized middleware requirements in [5] and we adopt and slightly modify them as follows.

- **Scalability**: The number of context consumers and context providers can grow very large. Nevertheless, collection and storage of context information must be efficient and context consumer's requests for context information should be processed efficiently.
- **Security**: Security for context information stored in context providers and context information exchanged among consumers and providers must be guaranteed. Authenticity of participants in the context information exchange should be proven and the integrity of the exchanged context information must be guaranteed. There should be proper access control mechanisms so that context information should be disclosed only to authorized consumers in a proper granularity. Moreover, there must be proper information flow control mechanisms so that the information flow of context information among providers and consumers should be controlled and, if necessary, traced.
- **Heterogeneity**: Not only various types of sensor devices but also many different kinds of context consumer hardware such as PDAs, notebooks, desktop computers, etc must be supported. We have to consider a variety of operating systems, programming languages, and interfaces, too.
- **Ease of configuration**: Hardware and software components providing contexts should be easily configured into the ubiquitous system and context consumer components must also be easily connected to the system.
- **Mobility**: Context consumers and providers can change their location. In spite of their movements, their location should be identified and proper routes to them should be found so that context-related messages can be delivered to them correctly.
- **Intelligence**: Sensors usually provide only low-level basic context information. But users may want to make decisions based on high-level knowledge inferred from basic information. The middleware should provide intelligence mechanism bridging these semantic gaps between sensors and users.

3 Architecture of the Middleware

In this section we first describe the overall architecture of the proposed middleware. Figure 1 shows the overall architecture along with its environment. Conceptually it consists of two components: context consumer component and context provider component. A user who wants to get context information registers its request at the consumer context handler, which analyzes the context request and decides whether it needs the cooperation of context providers. If so, it decomposes the request into subtasks and sends those subtasks that should be processed by outside context providers to the proper context providers. It can find the IP addresses of context providers by consulting either context brokers or DNS servers. While processing a context request, the consumer context handler can access consumer knowledgebase to make intelligent decisions. The knowledge is provided by users.

There are two kinds of entities collecting and providing context information. The first kind is a sensor network which is deployed at some area and collects context information from that area. There can be a large number of sensor networks, each of which is responsible for some geographical area and some sensor types. A sensor network consists of one coordinator and a large number of sensors. The context provider component in Figure 1 resides in the coordinator. When a certain sensor network is deployed, its coordinator registers the following information at the context broker: the IP address of the coordinator and the area and the sensor types. The second kind of context providers is any type of computing nodes such as PDA or PCs. The context provider component deployed on a node collects and stores context information pertaining to the node itself or the owner of the node. This node registers its IP address at the DNS servers.

The context provider provides two kinds of contexts: dynamic contexts and static contexts. User locations

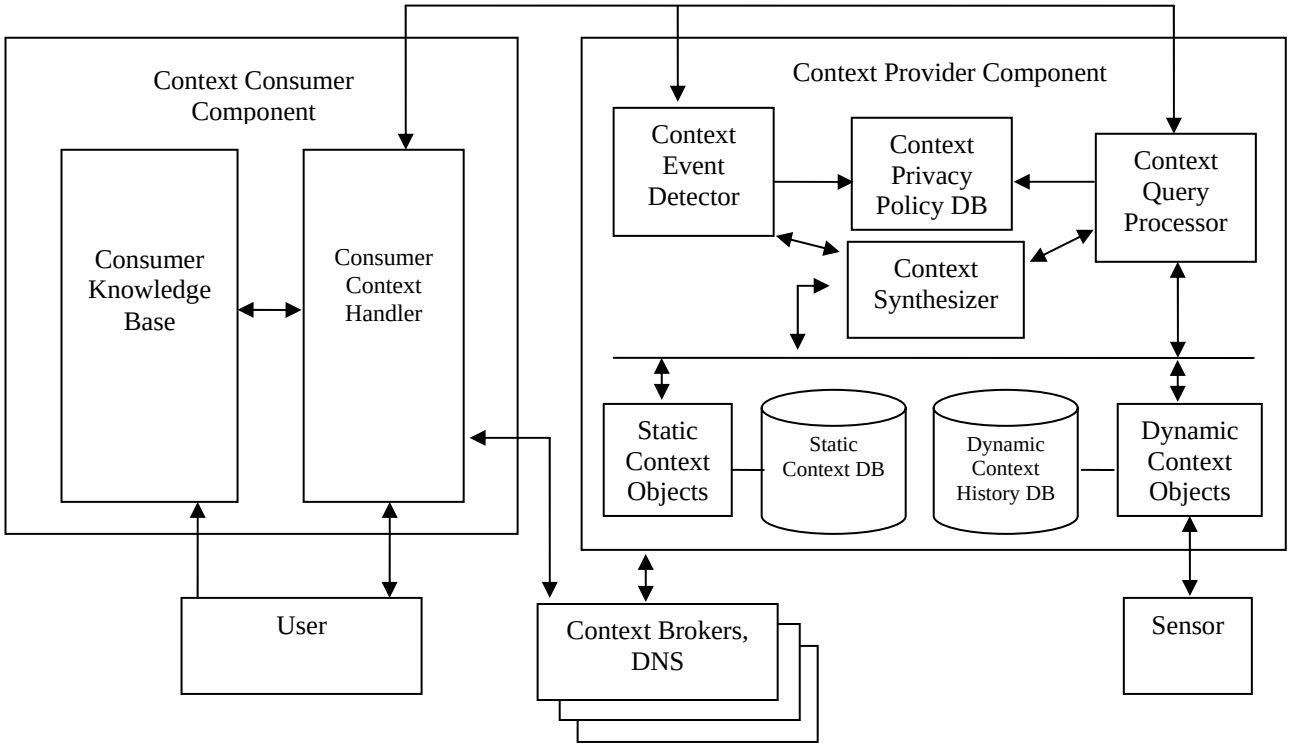


Fig.1 Proposed Architecture for Middleware

and temperatures are examples of dynamic contexts that can change over time and are collected by sensors. Contexts such as building layout and a person's phone number are fairly static and stored in a database. A dynamic context object provides object-oriented abstraction of a sensor collecting dynamic context information while a static context object provides the object-oriented abstraction of static context information. These context objects provide only low-level context information and the context synthesizer is introduced to enable high-level contexts to be specified and then inferred using the low level contexts.

Context consumers can obtain context by sending queries to context providers. The query is answered by the context query handler using context information in the dynamic/static context objects and the context synthesizer. Context synthesizers infer complex context information from basic context information and rules. A context consumer can also request to be notified when a context that they are interested in becomes true by registering the context specification to the context event handler, which detects when that context becomes true and notifies the consumer.

When providing and obtaining context, we have to consider two security issues: integrity and privacy. When obtaining contexts. A context consumer expects that contexts come from only authentic context providers and their contents are not modified by intruders and, therefore, wants the integrity of contexts. Likewise, context providers also want the integrity of context request messages from context consumers. Privacy of contexts is an important issue, too. Context information should not be disclosed to unauthorized entities. Moreover, context information should be provided only at a proper resolution level. For example, when providing location information of a certain person, the exact room number in which that person is located can be provided to some entities while only the building number should be provided to other entities. Context privacy policies dictate who can get context information at what resolution level.

4 Supports for Some Requirements

In this section we explain how the proposed architecture supports requirements for the middleware for ubiquitous computing.

4.1 Support for Scalability

One of the most important issues in enhancing scalability is how to efficiently handle requests for queries and event detections. To achieve this goal, it is desirable to decompose the specification of queries and events into subtasks and distribute them to context providers. Here we only consider context providers consisting of sensor networks for brevity. For the purpose of explanation, we assume that the whole area is divided into organizations, an organization consists of buildings, and a building is comprised of rooms. Location names are specified hierarchically like Internet domain names. So rm707.buildingT.hongik is the room 707 at the building T in the Hongik University. Any suffix of this name can be used as location names. Location names can also include variables. So ?x.hongik means some unknown building in Hongik University. We also assume that each sensor network is responsible for a building and collects data of a certain type from that building.

We first explain algorithms for decomposing and allocating query specifications. We consider following two query specifications. These specifications can be depicted as trees as in the figure 2.

```
Q1 = (and (location (person john)
                (locationName ?x.?y.hongik))
          (location (noOfPeople > 10)
                (locationName ?x.?y.hongik)))
Q2 = (or (location (person john)
                (locationName buildingA.hongik))
         (location (person john)
                (locationName buildingB.hongik)))
```

Q1 queries if John is in a certain room and there are more than 10 people in that room. In this case the whole query specification should be distributed to the coordinators of all the buildings in the Hongik

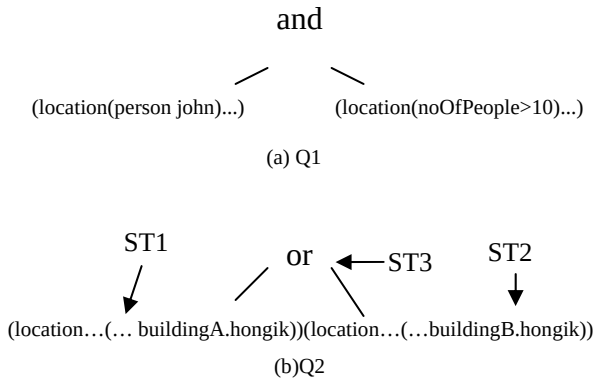


Fig.2 Specification Trees for constraints

University. Q2 asks if John is in either the building A or B. The first subtask (ST1) of determining if John is in the building A is allocated to the building A's coordinator and the second subtask (ST2) of determining if John is in the building B is allocated to the building B's coordinator. The subtask (ST3) of combining results of these two subtasks is the responsibility of the consumer context handler from which the original query was sent.

During the execution of the decomposition algorithm, a location information tree (LIT) is built as in the figure 3. For each query specification node there is one LIT node consisting of the unit field and the range field. The unit field specifies whether the corresponding specification node will be allocated to coordinators or the consumer context handler. If the specification node is to be allocated to coordinators, it is allocated to all the coordinators in the area specified by the range field. However, if a query is location-independent, it can be answered at any place and, therefore, its LIT node has *don't care* for the unit field. The unit field can have following values.

- ***known building name***: spec. node is allocated to that building coordinator.
- ***location variable***: spec. node is allocated to all the building coordinators in the area specified by the range field.
- ***don't care***: the corresponding query specification is location-independent so it can be checked at any node.
- ***consumer***: spec. node is allocated to the consumer context handler.

An LIT is built using the following algorithm.

```
decompose (spec. node) {
  if (leaf node)
    build-leaf-node-location-information-tree
      (spec. node)
  else /* inner node */
    if (node has a binary operator) {
      /* and, or */
      decompose (left child spec. node);
      decompose (right child spec. node);
      merge (left child spec. node's LIT,
             right child spec. node's LIT)}
    else /* node has a unary operator (not) */
      copy child spec. node's LIT}
```

The algorithm starts from the root node of a specification tree, goes down to the leaf node, and then

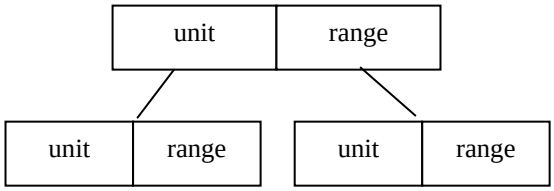


Fig.3 Location Information Tree

incrementally builds an LIT traversing the specification tree upward.

Build-leaf-node-location-information-tree builds an LIT for leaf nodes in a specification tree as follows.

(1) (location (person john)
(locationName buildingA.hongik))
→ (unit = buildingA.hongik, range = don't care)
(2) (location (person john)
(locationName ?x.?y.hongik))
→ (unit = ?y, range = hongik)
(3) (location (person john) (locationName hongik))
→ (unit = unknown, range = hongik)
(4) (weekday (date Today))
→ (unit = don't care, range = don't care)

The *and/or* operators have 2 specification subtrees as children and the *decompose* function is applied to these two subtrees, building one LIT for each of them. The *merge* function merges these two LITs by introducing a new LIT node having these two LIT as children. Its unit and range field values are determined as follows.

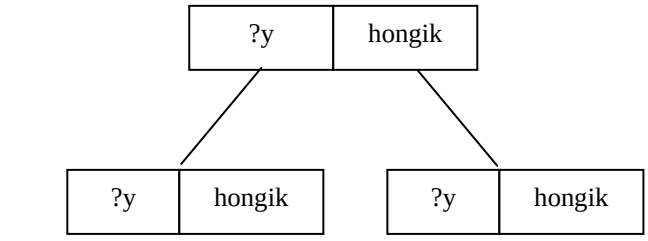
```
if (at least one child LIT has "don't care" in its
    unit field)
  copy the other child node's LIT node into the new
    LIT node which becomes the parent node
else if ((both child LIT nodes have the same known
          location name in their unit fields)
         or (both child LIT nodes have the same
          variable in their unit fields and the same
          content in their range fields))
  copy child spec. node's LIT node into the new
    LIT node
else
  put "consumer" value in the unit field of the new
    LIT nodes
```

Applying the above algorithm to two query specifications in the figure 2 results in LITs in the figure 4.

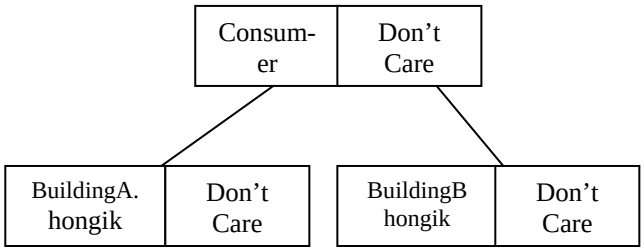
After the completion of the decomposition, the nodes in the specification tree are allocated to proper nodes using the following algorithm.

```
allocate(spec. tree) {
 if (spec. tree is null tree)
  return ()
 else /* spec. tree has at least one node */
    /* Lnode is the corresponding LIT node */
    /* for the spec. tree's root */
  if (Lnode's unit is "consumer" or "don't care") {
    allocate root spec. node to consumer context
      handler;
    if (operator of root of spec. tree is binary) {
      allocate(left spec. subtree);
      allocate(right spec. subtree)}
    else
      allocate(child spec. subtree)}
  else if (Lnode's unit is known building name)
    allocate the spec. tree to the designated
      building's coordinator
  else /* Lnode's unit is a variable */
    allocate spec. tree to coordinators of all
      buildings in area specified by Lnode's range}
```

The algorithm classifies specification tree nodes into three categories: (1) a node that should be allocated to a specific coordinator, (2) a node that should be allocated to all the coordinators in a certain area, and (3) a node that should be allocated to the consumer context handler. Then the algorithm allocates the



(a)LIT for Q1



(b)LIT forQC2

Fig.4 Resulting Location Information Trees

specification nodes to proper coordinators or the consumer context handler. Using this algorithm and the location information trees in the figure 4, the whole query Q1 in the figure 2 is allocated to the coordinators of all the buildings in the Hongik University while ST1, ST2, and ST3 of the query Q2 are allocated to the building A's coordinator, the building B's coordinator, and the consumer context handler, respectively.

For the decomposition and allocation of event specification, the same algorithms are used with only minor changes.

4.2 Support for Security

To guarantee the integrity of context requests and replies, we assume that all the users and context providers have public key and private key pairs. These key pairs are generated and distributed by a certificate authority, whose public key is known to all. An entity's public key is published as a certificate signed by the certificate authority and an entity's certificate has not only the entity's id and public key but also its role. When an entity sends a context request to the context provider, it sends a packet containing (request, requestor-id, timestamp, sign, certificate). The sign is calculated over (request, requestor-id, timestamp) using the private key of the requestor.

The signed context request is received by a proper context provider, which first verifies the integrity of the packet and checks the role of the requestor using the sign and the certificate.

There can be two kinds of basic context requests. The first includes a user name and the other does not. A basic request of the first kind asks questions on contexts pertaining to a particular person and they can be answered by the context owner. Some examples of these requests are as follows.

```
(location (person john) (locationName ?x))
(location (person john)
   (locationName room707.buildingT.hongik))
```

The first query asks the location of John and the second asks whether John is in a particular room. They can be answered by the node representing John. This node becomes the context provider and has privacy policies for context requests of this type and these policies are called per-user context privacy policies. Although we just used the name John, in reality the name can be in the form of an e-mail address and we can find the address of John's node with the help of the DNS service. An example of per-user context privacy policies for John is as follows.

Context Type	Requestor ID/Role	Resolution
Location	Mary	Building
Location	Don	Room

The resolution level for context requests of the location context can be a room (lowest resolution), a building, or an organization (highest resolution). In the above example Mary can ask questions at the building level while Don can ask questions at the room level. Let's assume that John is at the room 707 of the building T in the Hongik university. If Mary sends the above two context requests, the context provider will return buildingT.hongik as an answer to the first request but will return DN (Don't Know) to the second request because the allowed resolution level for Mary is a building but the request was posed with the resolution level of a room. If Don sends the same context requests, he will get room707.buildingT.hongik and Yes as an answer, respectively.

The second type of context requests does not ask questions on contexts belonging to a particular person. Some examples are as follows.

```
(temperature (location buildingT) (value ?x))
(temperature (location buildingT) (value 27))
```

Sensor network coordinators become context providers and have per-sensor network context privacy policies. The following table includes example per-sensor network context privacy policies applying to the above requests.

Context Type	Requestor ID/Role	Resolution
Temperature	Mary	5
Temperature	Don	1

The resolution level for this temperature context type can be specified as a real number such as 1, 2.5, 5, etc. We assume that the temperature of the building T is 27. If Don sends above context requests, he will get 27 and Yes as answers. But if Mary sends the same context requests, the context provider will return (**and** 25<= <30) to the first request and DN (Don't Know) to the second request.

The answers returned will also be signed with the private key of a context provider and will be in the form of either Yes with some value, No, or DN. Basic context requests can be combined into a composite context request using operator such as **and**, **or**, **not**. When partial answers are combined to answer a more complex context request, the following rules apply.

For an operator of the **and** type:
(**and** Yes DN) = DN, (**and** No DN) = No
For an operator of the **or** type:
(**or** Yes DN) = Yes, (**or** No DN) = DN
For the **not** operator:
(**not** DN) = DN

Entries in the privacy policies include the following additional fields to support traceability and control information flows.

- **Logging**: if yes, whenever the corresponding context information is accessed, the identity of the requestor, the access time, and the provided value are recorded to support the traceability.
- **Retention period**: specifies how long the requestor can retain the provided context information.
- **Forwardibility**: if yes, the requestor can forward the provided context information to some other entity.

4.3 Supports for other Requirements

The proposed middleware supports other requirements as follows.

- **Heterogeneity and ease of configuration**: Variety of sensors and nodes providing context information are modeled as dynamic context objects using a standard format. Moreover, context consumers access context information using standard communication protocols and message formats. Application programs can be connected to the context consumer components of the middleware through standard programming interfaces. All these standard features on data modeling, communication protocols, message formats, and programming interfaces help deploy and configure heterogeneous sensors and user application software and hardware without incurring much difficulty.
- **Mobility**: Mobility of a single node acting as a context consumer or context provider can be supported by the standard mobile IP mechanism easily. When the location and/or range of a sensor network changes, this change is detected and reported to the broker system by the coordinator of the moving sensor network. Therefore, context consumers need not be aware of the movement of sensor networks.
- **Intelligence**: Intelligence mechanisms are supported at two places. Context provider has context synthesizer to infer high-level context information from basic context information. Context consumer has consumer knowledge base and inference mechanism in the consumer context handler to enable users to arrive at high level decisions using context information from context providers.

5 Conclusion

Middleware for context-aware ubiquitous computing systems should provide basic services for discovering proper context providers, communicating context request/reply messages, processing context requests, collecting/storing context information, and making high-level decisions using basic context information and knowledge. In this paper, we described the architecture for middleware providing these basic services. We explained how the proposed middleware architecture supported scalability through the distributed processing of context requests and supported security by guaranteeing the integrity and privacy of context information itself and messages carrying it. We also briefly explained how the middleware could handle issues such as heterogeneity, ease of configuration, mobility, and intelligence.

References:

[1] Dey, A.K., Salber, D., Abowd, G.D., *A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications*, Human-Computer Interaction 16, 2001, pp. 97-106
[2] Roman, M. et al, *Gaia: A Middleware Infrastructure for Active Spaces*, IEEE Pervasive Computing, Special Issue on Wearable Computing 1, 2002, pp. 74-83
[3] Chen, G. and Kotz, D., *Design and Implementation of a Large-Scale Context Fusion Network*, Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services, 2004
[4] Hong, J.I. and Landay, J.A., *An Architecture for Privacy-Sensitive Ubiquitous Computing*, MobiSys, 2004, pp.177-189
[5] Henricksen, K. and Indulska, J., *Middleware for Distributed Context-Aware Systems*, LNCS 3760, 2005, pp. 846-863