Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 519 – 526 © 2006 PIPS

ISSN 1896-7094

Secure Distributed Processing of Context-Aware Authorization *

Young-Chul Shim

Hongik University, Department of Computer Engineering
72-1 Sangsudong, Mapogu, Seoul, Korea
shim@cs.hongik.ac.kr

Abstract. In this paper we present a framework for context-aware authorization in ubiquitous computing environments. We present an architecture consisting of authorization infrastructure and context infrastructure. The context infrastructure provides context information and the authorization infrastructure makes decisions to grant access rights based on context-aware authorization policies and context information. This paper also describes how multiple nodes in distributed environments cooperate to perform evaluation and detection of context constraints and events included in authorization policies while the integrity and privacy of context information are guaranteed.

1. Introduction

In this paper we present a framework for context-aware authorization in ubiquitous computing environments. It includes an infrastructure for context-aware authorization that consists of an authorization infrastructure and a context infrastructure. The former allows the enforcement of authorization policies based on context information while the latter provides context information. Context users obtain contexts by submitting queries or requesting the detection and notification of events. The framework also includes a context-aware authorization policy specification language with which one can authorize/prohibit access requests, initiate/terminate management actions, and delegate/revoke access rights.

Processing of authorization policies requires evaluation of context constraints and detection of context events. Because context information is collected by a large number of distributed nodes, the task of constraint evaluation and event detection requires the cooperation of distributed nodes. In this paper we present how the specification of constraints and events can be decomposed and allocated to distributed nodes so that they can evaluate constraints and detect events in a collaborative way. During distributed processing of constraints/events using context information, we want to ensure the integrity of the messages to and from the context infrastructure so that the messages may not be unlawfully modified or fabricated. We also want context information to be disclosed to only authorized entities at the proper resolution level and, therefore, want to maintain the privacy of context information. In this paper we also address these security issues in the distributed context processing.

Many researchers have studied a context infrastructure for context-aware applications [1, 2]. Within a Gaia project, an infrastructure for context-aware security services, called Cerberus, was proposed [3]. Its authorization infrastructure is similar to ours but its context infrastructure is primitive and the interaction between these two infrastructures is not clearly explained. There have been many studies on policy specification in security and network management areas but most of them are either too narrowscoped or too complex for practical uses. One of the most comprehensive and practical policy languages is Ponder [4]. We base our policy language on Ponder and extend it so that the context constraints and the interaction with the context infrastructure can be specified. Two basic approaches for including contexts in policy specification is either representing them as constraints in first order logic [5] or capturing them as roles in RBAC [6]. We take the first approach because it is more expressive. Finally we note that there has been little research on distributed processing of context-aware authorization policies and guaranteeing integrity and privacy of context information except a few works [7].

* 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

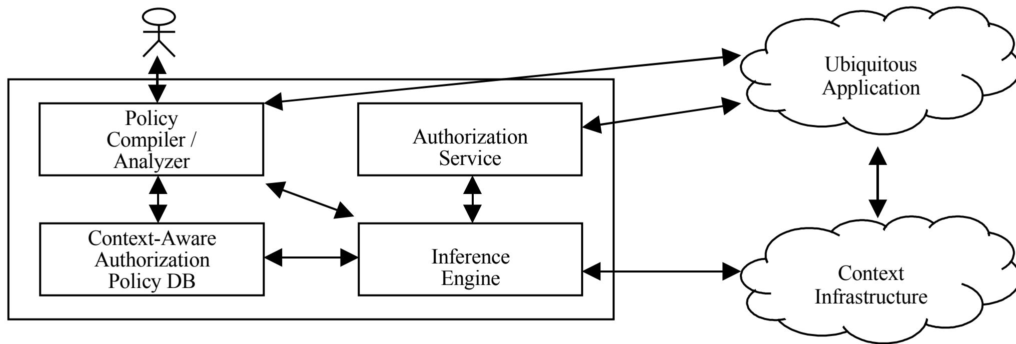

Fig. 1 Authorization Infrastructure Overview

The rest of the paper is organized as follows. Section 2 describes authorization and context infrastructures and Section 3 presents the authorization policy specification language. Section 4 presents algorithms that decompose the constraint/event processing task and allocate to distributed nodes. Section 5 explains how to address integrity and privacy issues and is followed by the conclusion in Section 6.

2. Context-Aware Authorization Infrastructure

Figure 1 shows the overall architecture of the authorization infrastructure. A ubiquitous application that wants to access a resource sends its access request to the authorization service module to get the authorization. The authorization service module consults the inference engine to determine whether to permit the access request and returns the result to the application. The inference engine makes its decision based on authorization policies stored in the policy database. Because authorization policies are written in terms of not only information on the request itself but also context information, the inference engine consults the context infrastructure to retrieve context information.

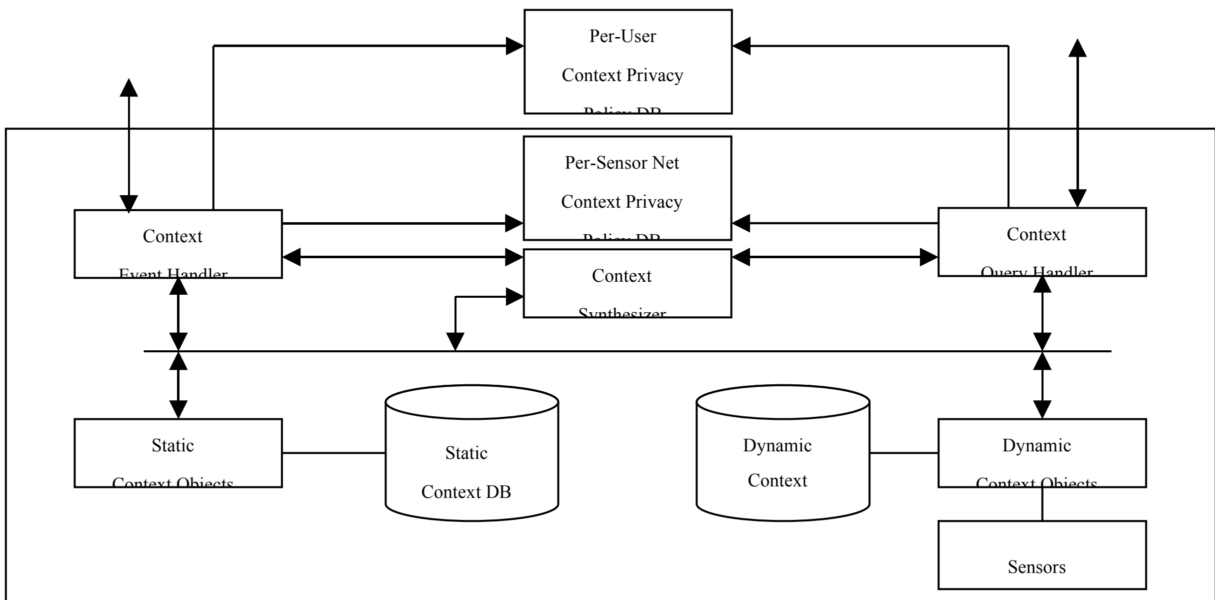

Fig. 2 Context Infrastructure Overview

Figure 2 shows the overall architecture of the context infrastructure. The infrastructure provides two kinds of contexts: dynamic contexts and static contexts. User locations and temperatures are examples of dynamic contexts and are collected by sensors. Contexts such as a 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

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 contexts by sending queries. The query is answered by the context query handler using context information in the dynamic/static context objects and the context synthesizer. 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.

When providing and obtaining context, we have to consider two security issues: integrity and privacy. 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 also an important issue. 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 others. Context privacy policies dictate who can get context information at what resolution level. Context privacy policies are stored in databases of two types: per user and per sensor net context privacy policy databases. Privacy policies on contexts such as a user location should be controlled by that user and are stored in the former database while privacy policies on contexts such the temperature of a location need to be controlled by the coordinator of the sensor network that measures that location's temperature and are stored in the latter database.

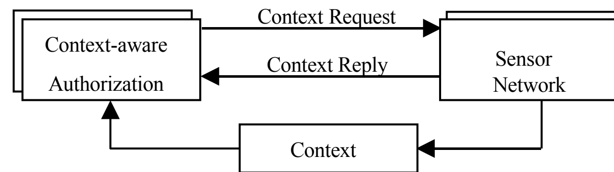

Fig. 3 Relation between Context-Aware Authorization Services and Sensor Networks

The whole context infrastructure consists of 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. All the function modules and databases of a context infrastructure in the figure 2, except the sensors and per user context privacy policy databases, reside in the sensor network coordinator. Each sensor network becomes a context provider while context-aware authorization service becomes a context consumer and the relationship between these two entities is depicted in figure 3. 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 which the coordinator is responsible for. When a context-aware authorization service needs context information, it first consults the context broker to get the IP addresses of the proper sensor network coordinators and visits those coordinators.

3. Authorization Policy Specification

There are 6 basic policy types in the proposed language. They are **authorize**, **prohibit**, **initiate**, **terminate**, **delegate**, and **revoke**.

An **authorize** policy allows a subject to perform operations on a target and a **prohibit** policy forbids a subject to perform operations on a target. Their syntaxes are as follows.

policy (**authorize** | **prohibit**) policy-Name "{"
subject object-Expression ;
target object-Expression ;
operation operation-Expression ;
[**when** constraint-Expression ;]
[**while** constraint-Expression ;]
[**entry** action-Expression ;]
[**exit** action-Expression ;] "}"

We explain the above syntax with an **authorize** policy. When an authorization request is made, the constraint expressions in the **when** and **while** clauses are evaluated and the requested operations are permitted only if both evaluation results are true. The constraint expression in the **while** clause should remain true, while the authorized operations are being executed. Otherwise the operations should be terminated. When the autho-

rization request is made, actions in the **entry** action clause are executed. When the authorized operation is completed, actions in the **exit** action clause are executed.

An **initiate** policy dictates that a subject should perform operations on a target while a **terminate** policy specifies that operations that a subject is performing on a target should be terminated. These policies are activated when the occurrence of the specified event is detected and their syntaxes are as follows.

policy (**initiate** | **terminate**) policy-Name [reuse-Flag] "{"
[**subject** object-Expression ;]
target object-Expression ;
operation operation-Expression ;
on event-Expression ;
when constraint-Expression "}"

When the event in the **on** clause is detected and the constraint in the **when** clause is satisfied, the **initiate** / **terminate** policy is activated. We omit the detailed description of delegate/revoke policies in this paper.

As we can see in the above, policy specifications can include specifications of context constraints and context events and we explain about them. A basic context constraint is specified using the context object definition and its syntax is as follows.

"(" context-Object-Name { "(" attribute-Name attribute-Expression ")" } ")"

An attribute expression can be either a constant such as a number or a string, a variable, or a relational expression. Followings are examples of basic context constraints.

(location (person kim) (locationName rm707))
(temperature (tempValue > 40C) (locationName ?X))

The first constraint becomes true when a person named kim is in the room 707. The second constraint becomes true if there is a room with its temperature higher than 40C. ?X is a variable. If the constraint is evaluated to be true, the variable returns the names of the rooms satisfying the condition.

The policy specification language permits complex constraints to be composed from basic context constraints using following logical operators: **and** , **or** , **not** .

- *"(" **and** context1 context2 shared-Variable-Constraint ")"* : becomes true when context1 and context2 are true and the variable constraint is true. A shared variable constraint is a Boolean expression on the shared variables in context1 and context2.
- *"(" **or** context1 context2 ")"* : becomes true when either context1 or context2 is true.
- *"(" **not** context1 ")"* : becomes true when context1 is false.

A basic event is either the primitive event which is defined at a sensor and detected at that sensor at one time point or the change of a constraint expression value. The syntax for constraint change events is "(" **becomes** constraint-expression ")". Followings are some examples of basic events.

(enters (person kim) (locationName room707))
(becomes (temperature (locationName ?X) (tempValue > 40C)))

Complex events can be specified using operators defined as follows.

- *"(" **and** event1 event2 ")"* : both event1 and event2 occur.
- *"(" **or** event1 event2 ")"* : either event1 or event2 happens.
- *"(" **repeat** [?n] event1 ")"* : event1 occurs repeatedly. The optional variable ?n returns the number of event occurrences.
- *"(" **next** event1 event2 ")"* : event1 happens and then event2 happens.

4. Distributing Constraint/Event Processing Tasks

To enforce the authorization policies including context constraints and events, the inference engine should know whether constraints are satisfied and when events occur. To enhance the performance of constraint/event processing, it is desirable to decompose the specification of a constraint or event into subtasks and distribute them to sensor network coordinators. In this section we explain how a constraint/event specification can be decomposed and the task of processing this constraint/event is distributed to sensor network coordinators. 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 as follows 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 constraint specifications. We consider the following two constraint specifications. These specifications can be depicted as trees as in the figure 4.

C1 = (and (location (person john) (locationName ?x.?y.hongik))
(location (noOf People > 10) (locationName ?x.?y.hongik)))
C2 = (or (location (person john) (locationName buildingA.hongik))
(location (person john) (locationName buidlingB.hongik))

C1 is satisfied if John is in a certain room and there are more than 10 people in that room. In this case the whole constraint specification should be distributed to the coordinators of all the buildings in the Hongik University. C2 is satisfied 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 inference engine.

During the execution of the decomposition algorithm, a location information tree (LIT) is built as in the figure 5. For each 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 inference engine. 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 constraint is location-independent, it can be checked at any place and, therefore, its LIT node has *don't care* for its 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.
- ***unknown*** : spec. node is allocated to all the building coordinators in the area specified by the range field.
- ***don't care*** : the corresponding constraint specification is location-independent, so it can be checked at any node.

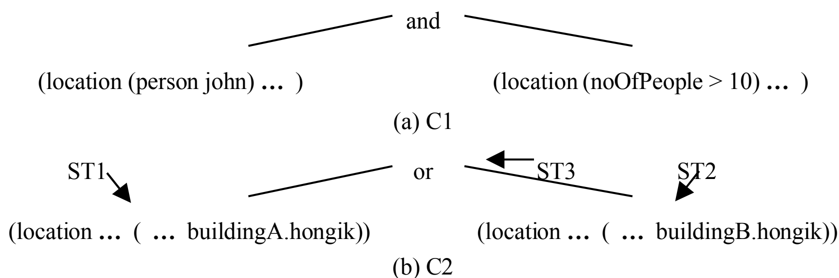

Fig. 4. Specification Trees for constraints

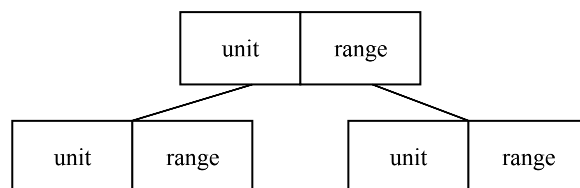

Fig. 5. Location Information Tree

- ***inference engine*** : spec. node is allocated to the inference engine.

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 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)

(a) LIT for C1 (b) LIT for C2

(3) (location (person john) (locationName hongik))
→ (unit = unknown, range = hongik)

(4) (weekday (date Today)) → (unit = don't care, range = don't care)

(5)

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 (parent) LIT 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 "inference engine" value in the unit field of the new LIT node
```

Applying the above algorithm to two constraint specifications in the figure 4 results in LITs in the figure 6.

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)
```

?y | hongik
?y | hongik ?y | hongik

Inference | Don't
BuildingA. | Don't BuildingB. | Don't

Fig. 6. Resulting Location Information Trees

```
      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 "inference engine" or "don't care") {
         allocate the root spec node to the inference engine;
         if (operator of the root of the spec. tree is binary operators)
            allocate(left spec. subtree); allocate(right spec. subtree)
         else
            allocate(child spec. subtree)
      else if (Lnode's unit is some known building name)
         allocate the spec. tree to the designated building's coordinator;
      else /* Lnode's unit is a variable */
         allocate the spec. tree to coordinators of all the buildings in the 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 inference engine. Then the algorithm allocates the specification nodes to proper coordinators or the inference engine. Using this algorithm and the location information trees in figure 6, the whole constraint C1 in figure 4 is allocated to the coordinators of all the buildings in the Hongik University while ST1, ST2, and ST3 of the constraint C2 are allocated to the building A's coordinator, the building B's coordinator, and the inference engine, respectively.

For the decomposition and allocation of event specification, the same algorithms are used. But because *and/or/next* are binary operators and *repeat* is a unary operator in event specification, the algorithms should be modified accordingly.

5. Securing the Distributed Processing of Context Requests

A request for context information can come from either an authorization infrastructure's inference engine or an ordinary application. To guarantee the integrity of context requests and replies, we assume that all the applications, users, inference engines, and sensor network coordinators 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. The role is set to 'infra' if the entity is either an inference engine or a coordinator. It is set to 'ordi' if the entity is an ordinary application or a user. When an entity sends a context request to the context infrastructure, it sends the 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 coordinator, which first verifies the integrity of the packet and checks the role of the requestor using the sign and the certificate. Privacy checking is needed during processing the context request only if the role of the requestor is 'ordi'.

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 is as follows.

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

These basic context requests query the location of John and John is the person who can determine the privacy policy for context requests of this type. Theses policies are called per-user context privacy policies and stored in the per-user context privacy policy database. Because the volume of total per-user context privacy policies will be huge, they will be stored at distributed nodes and we should be able to find the location of the node storing the particular policy. One way is that we deduce the node location from the name in the basic context request, John in the above example. 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 location of the node from this address with the help of the DNS service. An example of per-user context privacy policies for John is as follows.

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

The resolution level for context requests of the location sensor type is 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. We 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 infra 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 basic context requests does not involve a user name. Some examples are as follows.

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

The privacy policies for requests of this type are called per-sensor net context privacy policies and they are stored at the database of the corresponding sensor network coordinators. So for the above examples, the policies will be stored at the coordinator of the sensor network monitoring the temperature of the building T. Following is example per-sensor net context privacy policies applying to the above requests.

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

The resolution level for this temperature sensor 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 infrastructure will return (and 25<= <30) to the first request and DN (Don't Know) to the second request. If Mary wanted to get a Yes answer to the second request, she should have sent the following requests.

(temperature (location buildingT) (value (and 25<= <30)) or
(temperature (location buildingT) (value (and 20<= <30))

In general, if the resolution level is R, the values in the request or the answer should be (and R*n<= <R*(n+m)), where n and m are integers.

The answers returned will also be signed with the private key of a coordinator and they will be in the form of either Yes with some value, No, or DN. When partial answers are combined to answer a bigger context request at an inference engine, 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

6. Conclusion

In this paper we presented a framework for context-aware authorization in ubiquitous computing environment. The proposed framework consists of an authorization infrastructure and a context infrastructure. The former makes decisions to grant access rights based on context information and policies written in a flexible language while the latter provides contexts at various levels of abstraction. The policy specification language enables one to specify policies to authorize/prohibit access requests, initiate/terminate management actions, and delegate/revoke access rights. Specifications of context constraints and events are also included. To process authorization policies it is necessary for distributed nodes to cooperate during context constraint evaluation and context event detection. We explained how specifications of constraints and events can be decomposed and allocated to distributed nodes so that they can evaluate constraints and detect events in a collaborative way. We also proposed an approach to make the distributed context processing secure by guaranteeing the integrity of context requests and replies and the privacy of context information.

References

1. J. I. Hong & J. A. Landay, *An Infrastructure Approach to Context-Aware Computing,* HCI Journal **16** (2-3), 2001.
2. G. Judd & P. Steenkiste, *Providing Contextual Information to Pervasive Computing Application,* IEEE PerCom'03, 2003.
3. J. Al-Muhtadi et al, Cerberus: *A Context-Aware Security Scheme for Smart Spaces,* IEEE PerCom'03, 2003.
4. N. Damianou et al, *The Ponder Policy Specification Language,* LNCS 1995, 2001.
5. A. Ranganathan & R.H. Campbell, *An infrastructure for context-Awareness based on First Order Logic,* Pers. Ubiquit. Computing, 7, 2003, 353-364.
6. M. J. Covington et al, *Securing Context-Aware Applications using Environment Roles,* SACMAT'01, 2001, 10-20.
7. K. Minami & D. Kotz, *Secure Context-Sensitive Authorization,* Pervasive and Mobile Computing, 1, 2005, 123-156.