A Distributed Middleware Solution for Context Awareness in Ubiquitous Systems

Saad Liaquat Kiani, Maria Riaz, Yonil Zhung, Sungyoung Lee, Young-Koo Lee
Real Time and Multimedia Lab, Kyung Hee University, Kihung, Yongin, 449-701, South Korea
{saad, maria, zhungs, sylee, yklee}@oslab.khu.ac.kr

Abstract

Context aware middleware infrastructures have traditionally been implemented with a modular approach to allow different components to work cooperatively and supply context synthesis and provision services. In this paper, we discuss the important requirements that arise when such a middleware is deployed in a distributed environment and present the design and implementation of Context Aware Middleware for Ubiquitous Systems (CAMUS[1]) with which the authors have attempted to meet those requirements. Issues related to distributed coordination within the middleware in terms of component discovery and management and multiple context domains are also discussed.

1. Introduction

Different approaches have been proposed for building context-aware applications and services. Anind Dey et al [1] have built a Context Toolkit to support rapid prototyping of certain types of context-aware applications by providing a number of reusable components. Besides the Toolkit approach ([2], [3]), middleware infrastructures have been proposed [4], [5], [6] encompassing uniform abstractions and reliable services for common operations, support for most of the tasks involved in dealing with contexts, and thus simplify the development of context-aware applications.

The authors have proposed a *Context Aware Middleware for Ubiquitous Computing* (CAMUS) to address the discussed issues. The details of CAMUS middleware architecture with respect to context synthesis and data procurement have been adequately discussed in [7]; in this paper we will focus our discussion of CAMUS towards the distributed nature of the infrastructure, coordination and management aspects. Sec. 2 presents the overview of the CAMUS and is diagrammatically depicted in Fig. 1. Sec. 3 lists the issues which render a distributed middleware approach necessary for implementation and deployment of the middleware. The design considerations for the coordination framework which lead to a service oriented design are explained in Sec. 4. Prototype implementation's overview is given in Sec. 5 and the discussion is concluded in Sec. 6.

[1] This work is supported in part by the Korea Science and Engineering Foundation (KOSEF)

2. Functional Overview of CAMUS

The CAMUS architecture is provides context aware services after undergoing four steps including sensor access, feature extraction, context synthesis and delivery. Fig. 1 shows the core CAMUS components that individually handle these tasks.

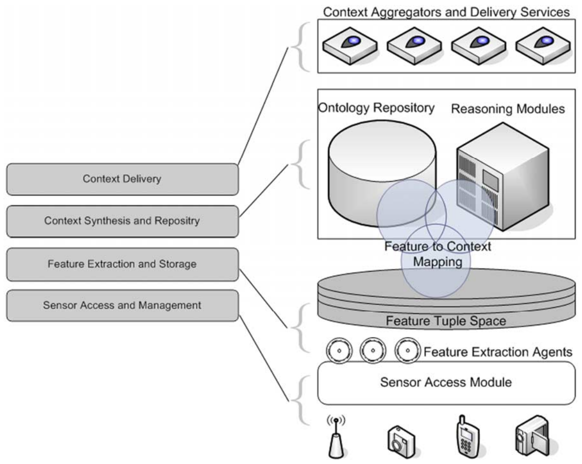
Context Aggregators and Delivery Services
Ontology Repository
Reasoning Modules
Context Delivery
Context Synthesis and Repositry
Feature Extraction and Storage
Sensor Access and Management
Feature to Context Mapping
Feature Tuple Space
Feature Extraction Agents
Sensor Access Module

Figure 1. CAMUS middleware architecture modules mapped into 4 separate functions

The lowest layer of CAMUS consists of a Sensor Access Module (SAM) which provides unified access to hardware sensors. SAM provides a Hardware

Abstraction Layer (HAL) which masks the heterogeneity of the environment sensors from the upper layers of the system. It provides a sensor driver API which allows native sensor drivers to be used to access the sensors in a unified manner.

Feature extraction agents (FXA) are sensing agents that extract the most descriptive features from the sensors through the SAM. The quantized features are encapsulated in the form of a 'Feature Tuple' and stored in Feature Tuple Spaces (FTS). Feature Tuple Space (FTS) is employed as underlying storage mechanism. Various sub-modules for feature extraction and context formation dynamically interact in the middleware by mere flow of objects in and out of the FTS. Current FTS implementation in CAMUS is done on top of IBM TSpaces [8] with extended read, write methods and customized events.

Feature - Context mapping layer performs the mapping required to convert a given feature into elementary context using reasoning mechanisms [7] and base on the meta-information saved in the ontology repository. Ontology Repository provides the basic storage services in a scalable and reliable fashion and contains the domain ontology (concepts and properties), contextual information (including both elementary and composite contexts), and meta-information. Reasoning Engine is a collection of various pluggable reasoning modules to handle the facts present in the repository as well as to produce composite contexts. Context Aggregator is responsible for satisfying certain context queries and providing context to interested applications through Context Delivery Services.

3. The Case for Distributed Middleware

Most of the current context aware systems that have been prototyped are limited to providing context at a small physical scale e.g. a campus environment [9], a laboratory, a home etc. At such a scale, the problems of a distributed environment do not present themselves adequately since the sensors are confined to a limited space (allowing easy sensor management), the context synthesis and delivery services run on a single system, system contact points are known [1] and dynamic discovery of system services is not required. In a practical scenario, an effective context aware system will provide context services over a vast stretch of environments ranging from homes, campuses, market places to city blocks and larger precincts. To manage such extended spaces it is necessary to separate the overall environment into smaller logical domains and incorporate a robust coordination and management framework as dictated by following reasons:

- The limitation in the communication range of most sensors makes it necessary for input gathering software components to exist in physical proximity to the sensors. Since sensors are diversely deployed in a ubiquitous environment, multiple input gathering components of a context aware system require coordination and management for data procurement.
- The complexity involved in context synthesis may require special computing devices for efficient performance e.g. a dedicate cluster of workstations set aside for context synthesis of all the entities registered with the system.
- Employing a single system to manage context synthesis and delivery for a large environment consisting of many sub-domains can be performance limiting. It is best that the whole environment is segregated into separate logical domains and clone sub-systems handle the steps involved in the context delivery process in each domain individually.

In order to carry out these varying specialized tasks and incorporate a considerably large number of hardware and software clients and contributors, a distributed setup becomes inevitable. In CAMUS, following requirements for coordination arise when functionality is distributed amongst components on specialized systems.

3.1. Logical and Physical Separation

The foreseeable problem that is related to heterogeneous sensors deployed in the environment is the limited communication range of sensors e.g. RFID, infrared, blue-tooth radio enabled sensors. It implies that the software components responsible for managing the sensors and retrieving measurements of environmental parameters have to be located in proximity to the sensors. With the distribution of sensors being wide and sparse, deployment of access components close to sensors becomes difficult and can by overcome by more than one access module to cover the set of sensors exhaustively and combining their results for context generation later on.

Since an entity's context is an interpreted result from a collection of features, it cannot be derived from a single sensory source. This constraint necessitates that such intermediate data is placed in storage till adequate information sources contribute and reasonable context can be inferred. In CAMUS, this underlying storage mechanism for data acquired from sensors is the FTS as discussed in Sec. 1 and provides a domain-wide persistent space. Instead of multiple sensor access modules storing the procured data in a single, central repository, it serves the performance requirements best that multiple localized repositories are used in conjunction with multiple sensors access modules. This not only reduces the communication delays but also provides a load balancing mechanism.

3.2. Context Domains

Ubiquitous computing environment is characterized by various domains e.g. home, office, university etc. To formally model context information to represent a particular domain, individual components need to be affiliated with a specific domain to relate coherent environments and entities, and to confine them within a logical boundary. Instead of employing a single context synthesis component to interpret context on behalf of all entities, the concept of separation of concerns based on geographical or logical boundaries has been implemented in CAMUS where individual domains are responsible for context management within domain boundaries as shown in Fig. 2.

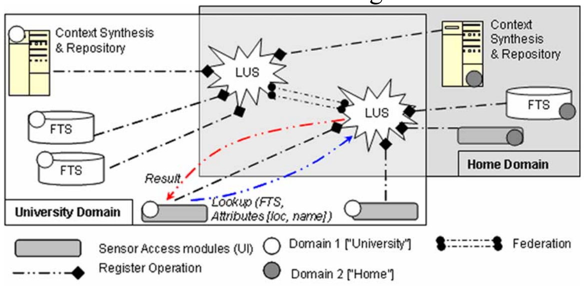
Context Synthesis & Repository
LUS
LUS
Context Synthesis & Repository
FTS
FTS
FTS
Home Domain
Result
Lookup (FTS, Attributes {loc, name})
University Domain
Sensor Access modules (UI)
Register Operation
Domain 1 ["University"]
Domain 2 ["Home"]
Federation

Figure 2. A scenario where components are associated with a single domain at a time

4. Service Oriented Approach to Middleware Coordination

CAMUS coordination sub-system is based on Service Oriented Architecture (SOA) [10] where core components are distributed services residing on the network to be published, discovered and invoked by each other. It allows a software programmer to model programming problems in terms of distributed services offered by components to anyone, anywhere over the network. To implement CAMUS architecture based on SOA, several existing technologies were investigated including Web Services, Java RMI, Jini [11], UPnP and CORBA; which are capable of, to one extent or another, satisfying the stated requirements. Web services and Jini are notable implementation of SOA and for reasons stated in Sec. 7, Jini technology was found to be the closest match to our requirements.

The discovery and registration process among modules is facilitated by Jini while the internal working of each module is independent of the underlying coordination mechanism.

4.1. Registration Mechanism and Formation of Domains

When a component becomes available, it joins a specific domain by registering the attributes and capabilities it affords along with a downloadable proxy with a service registry, specifically a Jini Lookup Service (LUS). Other components can discover the service by looking up specific attributes they are interested in. When the module of interest becomes available, its proxy is downloaded from the registry and is used for communicating back and forth between components.

The attributes published by components vary from basic properties such as name, service type, location, and status to specialized capabilities of the module for more specific lookup provision. Components can specify the location during lookup operations so they can locate other components (FTS in this case) in their proximity. Similarly, queries can be further restricted to domains memberships, e.g. an FTS belonging to 'University' domain may search only for SAM instances located in that domain.

4.2. Scalability

In a scenario where system users increase in number or the system has to manage a larger area equipped with a greater number of sensors, sudden changes and unpredictability in system configuration becomes inevitable leading to an increase in the responsibility of system components which may serve as a performance penalty. To meet this challenge at runtime, one of the solutions is to increase the number of system components handling the tasks that now offer an increased workload. In CAMUS, an increase in the number of sensors in a domain can be accommodated by the deployment of additional sensor access modules and/or feature tuple spaces as shown in Fig. 3. Efficient coordination amongst components requires that the increase in number of components does not hamper the discovery and registration process. For this purpose, multiple lookup services handling the task of discovery and registration are federated and the clients' (middleware services) queries are distributed across a number of lookup services to balance the load on the system and thus avoiding bottlenecks.

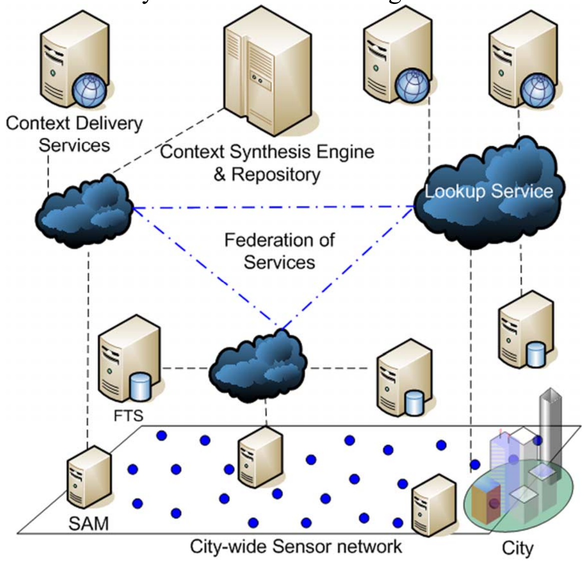

Figure 3. A hypothetical organization provides city wide context delivery service

5. Implementation Overview

The authors have implemented the CAMUS middleware infrastructure using Java and the coordination infrastructure is based on Jini technology. Individual components of CAMUS are deployed as services and their responsibility is limited to logical domains. After individual startup of core components, the middleware self configures itself to form a domain. A collection of drivers for sensors (audio, video (facial recognition), RFID tags and readers, Mote Kit sensors including light intensity, temperature and humidity sensors) and feature extraction agents have been made available as Java jar files. This collection of sensors and their respective feature extraction agents have been used to prototype a number of context aware applications, e.g. the well known meeting room scenario where presence of sufficient participants is detected by the system through sensor information, presentation material is distributed to their devices, actuators are used to dim room lights, presentation is projected and room temperature is kept to an optimal level.

6. Conclusion

The functionality of the components in the CAMUS system is independent of the discovery, registration and coordination scheme. A discovery and registration module is attached with each component which enables them to announce and locate each other. These services keep track of all the available service registries present in a given domain and facilitate the process of registration and discovery. This decoupling of the functionality from the communication scheme leverages flexibility to update or replace the components without affecting the communication infrastructure.

Jini was found to be the closest match to the specific requirements of our system. Particularly, the possibility of querying components by attributes, downloadable proxies and independence from transport protocol were the main support features which were found lagging in other similar technologies such as UPnP. Moreover, the advantages of leasing, remote and distributed event notification model and event mailboxes provided by Jini can be utilized to full extent in a distributed middleware architecture as CAMUS.

References

[1] Dey, A.K., et al.: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. In: Human-Computer Interaction (HCI) Journal, Vol. 16. (2001)

[2] S. Jang, Woo, W.: Ubi-UCAM: A Unified Context-Aware Application Model. In: Context 2003, Stanford, CA, USA. (Jun. 2003)

[3] Gellersen, H.W., Schmidt, A., Beigl, M.: Multi-Sensor Context-Awareness in Mobile Devices and Smart Artefacts. In: Mobile Networks and Applications, Vol. 7. (Oct. 2002) 341-351

[4] Hong, J.: The Context Fabric. http://guir.berkeley.edu/projects/confab/

[5] Ranganathan, A. and Campbell, R.H.: A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. In: ACM/IFIP/USENIX International Middleware Conference, Brazil. (Jun. 2003)

[6] Harry, C., Finin, T., and Joshi, A.: An Intelligent Broker for Context-Aware Systems. In: Ubicomp 2003, Seattle, Washington. (Oct. 2003)

[7] Ngo, H.Q., Shehzad, A., Kiani, S.L., Riaz, M., Lee, S.Y.: Developing Context-Aware Ubiquitous Computing Systems with a Unified Middleware Framework. In: Embedded and Ubiquitous Computing: EUC 2004, LNCS Volume 3207, Springer-Verlag 2004, pp. 672 – 681

[8] Wyckoff, P.: TSpaces, In: IBM Systems Journal, August 1998

[9] Burrell, J. and Gay, G.K. (2002). E-Graffiti: evaluating real-world use of a context-aware system. In: Interacting with Computers 14 (2002), 301-312.

[10] Furmento, N., Hau, J., Lee, W., Newhouse, S.: Darlington, J. Implementations of a Service-Oriented Architecture on top of Jini, JXTA and OGSA. In: Proceedings of the UK e-Science Program All Hands Meeting 2003, Nottingham, UK. Sept., 2003

[11] Sun Microsystems, Inc.: Jini^{TM} Architecture specification. http://www.sun.com/jini/specs/