Adaptive Software Deployment

Ichiro Satoh

National Institute of Informatics
2-1-2 Hitotsubashi Chiyoda-ku Tokyo 101-8430 Japan
Email: ichiro@nii.ac.jp

***Abstract*—Individual IoT devices may not be able to provide enriched services because they tend to have limited or imbalanced computational resources, e.g., lacking keyboards or displays. To solve this problem, we propose a dynamic federation of computational resources of smart objects as a virtual distributed system according to users' requirements and context. The approach presented in this paper has two key ideas. The first is to federate multiple smart objects or application-specific services as a virtual computer and the second is to separate between services for users from context-aware policies of such services. The approach was constructed as a general-purpose middleware system for executing and deploying application-specific software at smart objects.**

***Keywords*-Software deployment; Component coordination; Distributed system; Self-organization.**

I. Introduction

Internet of Things (IoT) has been used in a variety of infrastructures, such as power plants, energy distribution networks, transportation systems, water supply networks, and also the systems supporting the concept of smart houses, smart buildings and smart factories. Nevertheless, the bandwidth of networks between IoT devices tends to be narrow because such networks are based on low-power connections and radio communications. Furthermore, computational resources of IoT devices are not or well-balanced, in comparison with personal computers and smart phones. For example, IoT devices may lack any keyboards or displays. Although individual IoT devices do not have enough resources, their federations may be able to provide enriched services, which need computational resources, e.g., processors, memory, input/output devices beyond the resources of individual IoT devices.

To solve the problems discussed above, we propose to federate computational resources of IoT devices as a virtual distributed system. Such federations also should be adapted to contextual information, e.g., the locations of users and computers in addition to the computational resources of IoT devices. Our approach provides the notion of adaptive federations between the devices for software components running on IoT devices. It is characterized in introducing metaphors inspired from nature: *gravitational* and *repulsive* forces between software for defining services and target entities, including people or spaces that the services are provided for in the real world (Figure 1). This is because, like large-scale distributed systems, the scale and complexity of such a context-aware system is beyond our ability to manage using conventional approaches, such as centralized or top-down approaches. The former force dynamically deploys software for defining services at computers nearby the targets and executing them there. It is introduced as relocations between users and services or between services. The latter force prevents software for defining services from being at computers nearby the locations of the targets. It is used as a relocation technique between similar services.

Some of the metaphors in the approach were discussed in our previous paper [10]. In this paper, we address an application of the metaphors to a context-aware system in the real world. The approach is constructed as a general-purpose middleware system for federating IoT devices with the metaphors. To ensure independence from the underlying location systems, the system introduces virtual counterparts for the target entities and spaces. This paper presents the design and implementation of the system and our evaluation of our approach through a case study of it.

The remainder of this paper is organized as follows. In Section 2 we outline our basic idea behind the approach. Section 3 presents the design and implementation of the proposed approach. In Section 4 we describe our experience with the approach through an example. Section 5 surveys related work and Section 6 provides a summary.

II. Approach

This section discusses our requirements and then outlines idea behind the proposed approach.

A. Requirements

IoT devices are connected with one another via wired or wireless networks. They also tend to be various because they are often designed to their given purposes rather than any general-purposes.

- Individual IoT devices may not be able to provide enriched services because they tend to have only limited resources, e.g., lacking keyboards or screens.

- Since IoT devices are used in the real world, services provided from such devices often depend on contextual information. Nevertheless, the services themselves should be defined independently of context so that they can be reused in various context.
- Our middleware system should be independent of any underlying sensing systems to capture contextual information in the real world in addition on any application-specific services running on the system.
- IoT devices are often managed in a non-centralized manner to support large-scale context-aware services, e.g., city-level ones.

B. Adaptive federation of IoT devices

To satisfy the above requirements, we introduce the following approaches into our system.

- The first is to dynamically federate multiple IoT devices or application-specific services as a virtual computer. To satisfy the first requirement, the system dynamically makes a virtual computer on IoT systems by federating of hardware and software components according to their capabilities.
- The second is to separate between services provided for users from context-aware policies of such services. When contextual information changes, a federation among IoT devices should adapt itself to changes rather than individual components running on the devices.
- The system supports software or hardware components. It enables application-specific services to defined within such components rather than the system.
- It is constructed as a distributed system consisting of IoT devices, where IoT devices are managed in a peer-to-peer manner.

Context-aware services themselves are common. To reuse such services in other context, the middleware systems provides contextual conditions that the services should be activated as policies defined outside the services. The policies can be classified into two types: *gravitational* and *repulsive* forces between services and the target entities and spaces.

1) Virtual counterparts: We abstract away the underlying systems, including location-sensing systems. Our middleware system has the following two kinds of software components, called agents, in addition to hardware components corresponding to IoT devices.

- Physical entities, people, and spaces can have their digital representations, called *virtual-counterpart* agents, in the system. Each virtual-counterpart is automatically deployed at computers close to its target entity or person or within the space. For example, a virtual-counterpart for users can store per-user preferences and record user behavior, e.g., exhibits that they have looked at.

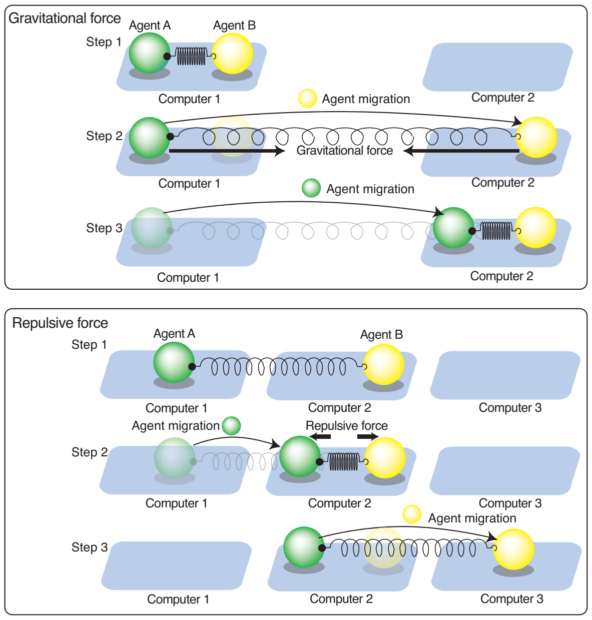


Figure 1. Component deployment based on *Gravitational* and *repulsive* policies

- The system assumes an application-specific service to be defined in a software component and executes the component as an autonomous entity, called a *service-provider* agent. In the current implementation, existing Java-based software components, e.g., JavaBeans, can define our services.

The first and second agent are executed in runtime systems and can be dynamically deployed at the runtime systems different computers. They are executed as mobile agents [12] that can travel from computer to computer under their own control. When a user approaches closer to an exhibit, our system detects that user migration by using location-sensing systems and then instructs that user's counterpart agent to migrate to a computer close to the exhibit.

2) Federation and deployment as forces between agents: As mentioned previously, we introduce nature-inspired agent deployment policies based on two metaphors: *gravitational* and *repulsive* forces. Virtual-counterpart and service-provider agents are loosely coupled so that they can be dynamically linked to others. The current implementation has two built-in *gravitational* policies:

- An agent has a *follow* policy for another agent. When the latter migrates to a computer or a location, the former migrates to the latter's destination computer or to a computer nearby.
- An agent has a *shift* policy for another agent. When the latter migrates to a computer or a location, the former migrates to the latter's source computer or to a computer nearby.

Each service-provider agent can have at most one *gravitational* policy. Although the *gravitational* policy itself does not distinguish between virtual-counterpart and service-provider agents, in the above policies, we often assume the former to be a service-provider agent and the latter to be a virtual-counterpart agent. For example, when a visitor stands in front of an exhibit, the underling location-sensing system detects the location of the visitor and then the visitor's virtual counterpart agent is deployed at a computer close to the current location. When service-provider agents declare follow policies for the counterpart agent, they are deployed at the computer or nearby computers.

The current implementation has two built-in *repulsive* policies. Each service-provider agent can have zero or more repulsive policies in addition to the *shift* policy.

- An agent has an *exclusive* policy for another agent. When the former and latter are running on the same computer or nearby computers, the former migrates to another computer.
- An agent has a *suspend* policy for another agent. When the former and the latter are running on the same computer or nearby computers, the former is suspended until the latter moves to another computer.

Each service-provider agent can have at the most one *repulsive* policy. To avoid the redundancy of agents whose services are similar at the same computers, we should use *repulsive* force between service-provider agents.

III. Design and Implementation

This section describes the design and implementation of our middleware system. As shown in Figure 2, it consists of two parts: (1) context information managers, (2) runtime systems for application-specific services. The first provides a layer of indirection between the underlying locating-sensing systems and agents. It manages one or more sensing systems to monitor contexts in the real world and provides neighboring runtime systems with up-to-date contextual information of its target entities, people, and places. The second is constructed as a distributed system consisting of multiple computers, including stationary terminals and users' mobile terminals, in addition to servers. Each runtime system runs on a computer in the real world and is responsible for executing and migrating virtual-counterpart and service-provider agents with nature-inspired deployment policies. It evaluates the deployment policies of agents and then deploys the agents at runtime systems. Application-specific services are defined as virtual-counterparts or service-provider agents, where the former offers application-specific content, which is attached to physical entities, people, and places, and the latter can be defined as conventional Java-based software components, e.g., JavaBeans.

A. Contextual Information Management

Each Context Information Manager (CIM) manages one or more sensing systems to monitor context in the real world,

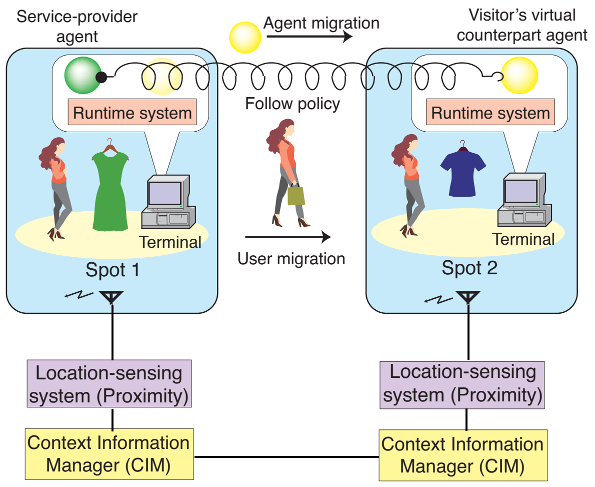

Service-provider agent
Agent migration
Visitor's virtual counterpart agent
Runtime system
Follow policy
Runtime system
Terminal
User migration
Terminal
Spot 1
Spot 2
Location-sensing system (Proximity)
Location-sensing system (Proximity)
Context Information Manager (CIM)
Context Information Manager (CIM)

Figure 2. Architecture.

e.g., people and the locations of the target entities and people. Among many kinds of contextual information on the real world, location is one of the most important and useful in managing services in IoT networks. Therefore, this paper focuses on sensing location of IoT devices and users although the manager itself can support a variety of context. The current implementation of CIM supports active Radio Frequency IDentifier (RFID)-tag systems to locate computers and users. CIM monitors the RFID-tag systems, detects the presence of tags attached to people and entities, and maintains up-to-date information on the identities of RFID tags that are within the zones of coverage of its RFID tag readers. To abstract away the differences between the underlying locating systems, each CIM maps low-level positional information from each of the locating systems into information in a symbolic model of the location. The current implementation represents an entity's location, called a *spot*, e.g., spaces of a few feet, which distinguishes one or more portions of a room or building. A CIM either polls its sensing systems or receives the events issued by the sensing systems or other CIMs. Each CIM has a database for mapping the identifiers of RFID tags and virtual counterparts corresponding to physical entities, people, and spaces attached to the tags. These database may maintain information on several tags. When a CIM detects the existence of a tag in a spot, it multicasts a message containing the identifier of the tag, the identifiers of virtual counterparts attached to the tag, and its own network address to nearby runtime systems.

B. Runtime system

Since services are defined as software components, the middleware systems provides runtime systems that can execute and migrate components to other runtime systems running on different computers through TCP channels using mobile-agent technology [12].

1) Execution and deployment of services: Each runtime system is built on Java virtual machine (Java VM) version 8 or later, which conceals differences between the platform architectures of the source and destination computers as shown in Figure 3. It governs all the agents inside it and maintains the life-cycle state of each agent. When the life-cycle state of an agent changes, e.g., when it is created, terminates, or migrates to another runtime system, its current runtime system issues specific events to the agent.

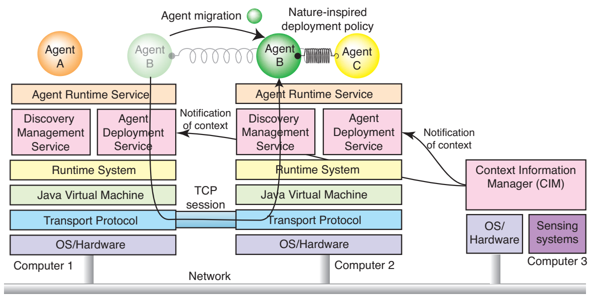


Figure 3. Runtime system

When an agent is transferred over the network, not only its code but also its state is transformed into a bitstream by using Java's object serialization package and then the bit stream is transferred to the destination. Since the package does not permit the stack frames of threads to be captured, when an agent is deployed at another computer, its runtime system propagates certain events to to instruct it to stop its active threads. Arriving agents may explicitly have to acquire various resources, e.g., video and sound, or release previously acquired resources.

The system only maintains per-user profile information within those agents that are bound to the user. It instructs the agents to move to appropriate runtime systems near the user in response to his/her movements. Thus, the agents do not leak profile information on their users to third parties and they can interact with mobile users in a personalized form that has been adapted to respective, individual users. The runtime system can encrypt agents to be encrypted before migrating them over a network and then decrypt them after they arrive at their destinations.

2) Policy-based federation and deployment: Our adaptive deployment policies are managed by runtime systems without a centralized management server. When a runtime system receives the identifiers of virtual counterparts corresponding to physical entities, people, and spaces attached to newly visiting tags, it discovers the locations of the virtual counterparts by exchanging query messages between nearby runtime systems. Each runtime system periodically advertises its address to the others through User Datagram Protocol (UDP) multicasting, and these runtime systems then return their addresses and capabilities to the runtime system through a TCP channel.

When an agent migrates to another agent's runtime system, each agent automatically registers its deployment policy with the destination. The destination sends a query message to the source of the visiting agent. There are two possible scenarios: the visiting agent has a policy for another agent or it is specified in another agent's policies. Since the source in the first scenario knows the runtime system running the target agent specified in the visiting agent's policy; it asks the runtime system to send the destination information about itself and about neighboring runtime systems that it knows, e.g., network addresses and capabilities. If the target runtime system has retained the proxy of a target agent that has migrated to another location, it forwards the message to the destination of the agent via the proxy. In the second scenario, the source multicasts a query message within current or neighboring sub-networks. If a runtime system has an agent whose policy specifies the visiting agent, it sends the destination information about itself and its neighboring runtime systems. The destination next instructs the visiting agent or its clone to migrate to one of the candidate destinations recommended by the target, because this platform treats every agent as an autonomous entity.

C. Service

Each mobile agent is attached to at most one visitor and maintains that visitors' preference information and programs to provide customized annotations. Each virtual counterpart agent keeps the identifier of the tag attached to its visitor. Each agent in the current implementation is a collection of Java objects in the standard JAR file format and can migrate from computer to computer and duplicate itself by using mobile agent technology. Each agent must be an instance of a subclass of the class pre-defined in the middleware system. Our system enables agents to define the computational resources they require. When an agent migrates to the destination according to its policy, if the destination cannot satisfy the requirements of the agent, the platform system recommends candidates that are runtime systems in the same network domain to the agent. If an agent declares repulsive policies in addition to a gravitational policy, the platform system detects the candidates using the latter's policy and then recommends final candidates to the agent using the former policy, assuming that the agent is in each of the detected candidates.

IV. Experience

To evaluate the performance overhead of the deployment policies presented in this paper, we implemented and evaluated a non-deployment policy version of the system. When this version detected the presence of a user at one of the spots, it directly deployed a service-provider agent instead of virtual counterpart agents. We measured the cost of migrating a null agent (a 5-KB agent, zip-compressed) and an annotation agent (1.2-MB agent, zip-compressed) from a source computer to a recommended destination computer that was recommended. The latency of discovering and instructing a virtual counterpart or service-provider agent attached to a tag after the CIM had detected the presence of the tag was 420 ms. Without any deployment policies, the respective cost of migrating the null and annotation agents between two runtime systems running on different computers over a TCP connection was 41 ms and 490 ms after instructing agents to migrate to the destination. When the null or annotation agent had a *follow* policy for the virtual counterpart agent, the respective cost of migrating the null and annotation agents between two runtime systems running on different computers over a TCP connection was 185 ms and 660 ms. These results demonstrate that the overhead of our deployment policy can be negligible in context-aware services.

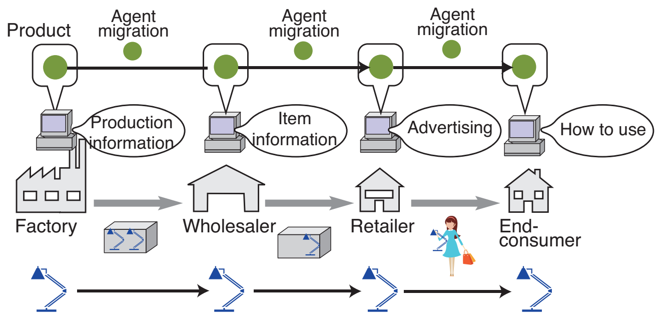


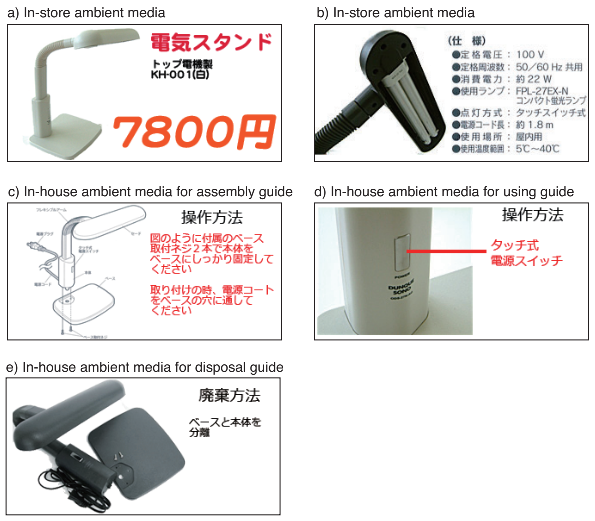
Figure 4. Forwarding agents to digital signage when user moves.

A. Advertisement media for appliances

We experimented and evaluated an advertisement system for appliances, e.g., electric lights, with the approach. It does not support advertising for its target appliance but also assist users to control and dispose the appliance. We attached an RFID tag to an electric light and provide a mobile agent as an ambient media for the light. As shown in Figure 4, it supports the lifecycle of the item from shipment, showcase, assembly, using, and disposal.

- **In warehouse:** While the light was in a warehouse, its counterpart object declared a *follow* policy to a services that should have been deployed and running at the computers of the warehouse's operators in the warehouse. The service displayed the item's specification, e.g., its product number, serial number, date of manufacture, size, and weight.
- **In a store's showcase:** While the light is being showcased in a store, its counterpart object declared two *follow* policies. The first policy was a relocation relation to an advertisement service fetched from the factory and the second policy was a relocation relation to price-tag service fetched from the store. The advertising service was deployed at a computer close its target item, which could display its advertising media to attract purchases by customers who visit the store. Two images, as shown in Figure 5 a) and b), are maintained in the agent that display the price, product number, and manufacturer's name on the current computer. The price-tag service communicated with a server provided by the store to know the selling price of its target, electric light and displayed the price on the display of its current computer.
- **In a store's checkout counter:**When a customer carried the item to the cashier of the store, the item's counterpart declared a *shift* policy to an order service. The service remained at a store to order another item for the factory as an additional order.
- **In house:** When a light was bought and transferred to the house of its buyer, its counterpart declared a *follow* policy to an instruction service at a computer in the house and provides instructions on how it should be assembled. The active media for advice on assembly are shown in Figure 5 c) and d). The service also advises how it was to be used as shown in When it was disposed of, the service presents its active media to give advice on disposal. As shown in Figure 5 e), the service provides an image to illustrate how the appliance is to be disposed of.

Our experiment at a store is a case study in our development of pervasive-computing services in large-scale public spaces. However, we could not evaluate the scalability of the system in the store, because it consisted of only four terminals. Even so, we have a positive impression on the availability of the system for large-scale public services. This is because the experimental system could be operated without any centralized management system. The number of agents running or waiting on a single computer was bound to the number of users in front of the computer.


a) In-store ambient media
電気スタンド
トップ電機製
KH-001(白)
7800円
b) In-store ambient media
(仕　様)
●定格電圧：100 V
●定格周波数：50／60 Hz 共用
●消費電力：約 22 W
●使用ランプ：FPL-27EX-N コンパクト蛍光ランプ
●点灯方式：タッチスイッチ式
●電源コード長：約 1.8 m
●使用場所：屋内用
●使用温度範囲：5℃～40℃
c) In-house ambient media for assembly guide
操作方法
図のように付属のベース取付ネジ２本で本体をベースにしっかり固定してください
取り付けの時、電源コードをベースの穴に通してください
d) In-house ambient media for using guide
操作方法
タッチ式
電源スイッチ
e) In-house ambient media for disposal guide
廃棄方法
ベースと本体を分離
Figure 5. Agents for appliance

V. Related Work

There have been many attempts to manage IoT devices as a distributed system [3][6][7][5]. Govoni, et al. [5] proposed a middleware system, called SPF, to support IoT application and service development, deployment, and management. However, SPF did not support any dynamic deployment and coordination between services and devices. Fortino et al. [6] proposed an agent-based middleware system for discovering IoT devices in IoT through a REST interface, for registering, indexing, and searching IoT devices and their events, but their system did not support any deployment and federation of software. Smart-Its [2] was a platform specifically designed for augmentation of everyday objects to empower objects with processing, context-awareness and communication instead of the deployment of services. Several researchers proposed the notion of disaggregated computing as an approach to dynamically composing between devices, e.g., displays, keyboards, and mice that are not attached to the same computer, into a virtual computer in a distributed computing environment. Leppaanen et al. [9] proposed a mobile agent-based middleware for IoT devices. Although it enabled software to be dynamically deployed at IoT devices, it lacked any mechanisms for federating software and devices.

That system presented in this paper is an application of our previous bio-inspired system [10]. The system was a general-purpose test-bed platform for implementing and evaluating bio-inspired approaches over real distributed systems. It enabled each software agent to be dynamically organized with other agents and deployed at computers according to its own organization and deployment policies. In contrast, this paper addressed a practical system with nature-based approaches used in the real world with real users for real applications. We presented an outline of mobile agent-based services in public museums in our earlier versions of this paper [11], but did not describe any nature-inspired deployment policies in those works.

VI. Conclusion

This paper presented a context-aware service middleware system with an adaptive and self-organizing approach. The system enabled two individual agents to specify one of the deployment policies as relocations between the agent and another. It can not only move individual agents but also a federation of agents over a distributed system in a self-organized manner. We evaluated the system by applying it to visitor-assistant services. When visitors move from exhibit to exhibit, the visitors' virtual counterpart agents can be dynamically deployed at computers close to the current exhibits to accompany the visitors via their virtual counterpart agents and play annotations about the exhibits. Visitors and service-provider agents are loosely coupled because the agents are dynamically linked to the virtual counterpart agents corresponding to them by using our deployment policies.

In conclusion, we would like to identify further issues that need to be resolved. We plan to evaluate existing bio-inspired approaches to distributed systems with the platform. We also plan to apply the system into other applications.

References

[1] O. Babaoglu, H. Meling, and A. Montresor, "Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems," Proceeding of 22th IEEE International Conference on Distributed Computing Systems, pp. 15-22, IEEE Computer Society, September 2002.

[2] M. Beigl and H. W. Gellersen, "Smart-Its: An Embedded Platform for Smart Objects," Proceedings of the smart object conference (SOC'2003), pp. 15–17, Springer, 2003.

[3] G. Bravos, "Enabling Smart Objects in Cities Towards Urban Sustainable Mobility-as-a-Service: A Capability – Driven Modeling Approach," Proceedings of International Conference on Smart Objects and Technologies for Social Good (GOODTECHS'2016), pp. 342-352, Springer, July 2016.

[4] B. L. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer, "EasyLiving: Technologies for Intelligent Environments," Proceedings of International Symposium on Handheld and Ubiquitous Computing, pp. 12-27, January 2000.

[5] M. Govoni, J. Michaelis, A. Morelli1, N. Suri, and M. Tortonesi, "Enabling Social- and Location-Aware IoT Applications in Smart Cities," Proceedings of International Conference on Smart Objects and Technologies for Social Good (GOODTECHS'2016), pp. 305-341, Springer, July 2016.

[6] G. Fortino, M. Lackovic, W. Russo, and P. Trunfio, "A Discovery Service for Smart Objects over an Agent-Based Middleware," International Conference on Internet and Distributed Computing Systems (IDCS'2013) pp. 281-293, LNCS, Vol.8223, Springer, October 2013.

[7] G. Fortino, A. Guerrieri, W. Russo, and C. Savaglio, "Middlewares for Smart Objects and Smart Environments: Overview and Comparison," in Internet of Things Based on Smart Objects pp. 1-27, Springer, 2014.

[8] B. Johanson, G. Hutchins, T. Winograd, and M. Stone, "PointRight: experience with flexible input redirection in interactive workspaces," in Proceedings of 15th ACM symposium on User interface software and technology, pp. 227-234, October 2002.

[9] T. Leppänen, J. Riekki, M. Liu, E. Harjula, and T. Ojala, "Mobile Agents-Based Smart Objects for the Internet of Things," In Internet of Things Based on Smart Objects: Technology, Middleware and Applications, (G. Fortino and P. Trunfio (ed.)), pp. 29-48, Springer 2014.

[10] I. Satoh, "Test-bed Platform for Bio-inspired Distributed Systems," in Proceesings of 3rd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS'2008), November 2008.

[11] I. Satoh, "A Context-aware Service Framework for Large-Scale Ambient Computing Environments," in Proceedings of ACM International Conference on Pervasive Services (ICPS'09), pp. 199-208, ACM Press, July 2009.

[12] I. Satoh, "Mobile Agents," Handbook of Ambient Intelligence and Smart Environments, pp. 771-791, Springer, 2010.

[13] P. Tandler, "The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments," Journal of Systems and Software, Vol.69, No.3, pp. 267-296, 2004.