Adaptive Deployment of Software in Smart-City

Ichiro Satoh
National Institute of Informatics
Tokyo
ichiro@nii.ac.jp

ABSTRACT

There are a variety of smart objects in smart cities, but their computational resources may not enough to provide enrich services for users. 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.

CCS CONCEPTS

• **Software and its engineering** → **Middleware**; • **Human-centered computing** → *Ambient intelligence*; • **Computing methodologies** → *Mobile agents*;

KEYWORDS

Middleware, Ambeint Intelligence, Mobile agent

ACM Reference Format:
Ichiro Satoh. 2018. Adaptive Deployment of Software in Smart-City. In *International Conference on Smart Objects and Technologies for Social Good (Goodtechs '18), November 28–30, 2018, Bologna, Italy,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3284869.3284886

1 INTRODUCTION

Smart cities are one of the most important and promising targets of pervasive or ubiquitous technologies. In addition to a variety of portable computing devices, e.g., smartphones, tablets, or smart watches combined, the emergence of many other small smart objects, e.g., digital signage and public terminals, with computational, sensing and communication capabilities is creating unprecedented opportunities for each of us to do something useful, ranging from a single person to communities in cities. These connected smart objects are finding their way into our pockets, vehicles, urban areas and infrastructure, thus becoming the very texture of our society and providing us the possibility, but also the responsibility, to shape it.

In fact, there are many kinds of smart objects in a smart city. Although such smart object are often equipped with 32-bit processors and 500 Mbytes or 1 Gbytes memory, e.g., Raspberry-Pi, their computational resources are not enough to provide enrich services for users. They tend to be designed for only their original purposes instead of any general ones. Individual smart objects may not be enough, their federations may be able to have resources to provide enrich services, which need computational resources, e.g., processors, memory, input/output devices beyond the resources of individual smart objects. For example, smart phones enable us to input information but lack any large size screen, which can be shared by multiple users. Digital signage may have large size screens but lack any input methods for users. We often need to federate computational resources of smart objects as a virtual distributed system. Such federations also should be adapted to contextual information, e.g., the locations of users and computers.

However, it is difficult to manage federations of smart objects because smart objects are dynamically added to or removed from the federations. Therefore, the scale and complexity of a federation of smart objects may be beyond our ability to organize and manage using conventional approaches such as centralized or top-down approaches. Nature-inspired approaches enable us to manage such a federation in a non-centralized and self-organizing manner. Our approach introduces two 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 (Fig. 1). The former dynamically deploys software for defining services at computers nearby the targets and executing them there. It is used as a relocation between users and services. The latter 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 [13]. On the other hand, this paper addresses an application of the metaphors into a federation of smart objects. The approach is also constructed as a general-purpose middleware system for federating smart objects by introducing the two above metaphors as a proof-of-concept of it. To ensure independence from the underlying location systems, the system introduces virtual counterparts for the target entities and spaces.

2 RELATED WORK

There have been many attempts to enabling smart objects to be used to achieve smart objects [5, 7–9, 16]. Govoni, et al. [7] 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Goodtechs '18, November 28–30, 2018, Bologna, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6581-9/18/11…$15.00
https://doi.org/10.1145/3284869.3284886

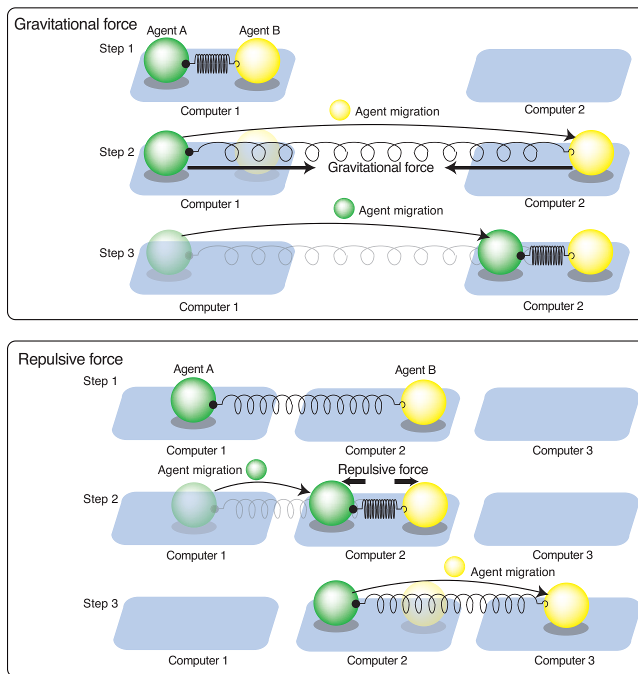

Figure 1: Gravitational and repulsive policies

services and devices. Fortino et al. [9] proposed an agent-based middleware system, named ACOSO, for discovering smart objects in IoT through a REST interface, for registering, indexing, and searching smart objects and their events, but their system did not support any deployment and federation of software. Smart-Its [4] 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. Voyager was a framework that supports the implementation of ambient dialogue applications to implement smart environments rather then any general-purpose 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. [11] proposed a mobile agent-based middleware for smart objects. Although it enabled software to be dynamically deployed at smart objects, it lacked any mechanisms for federating software and objects.

Several researchers have explored active media with the aim of enabling users to watch/listen to context-aware information, e.g., annotation about exhibits at the right time and in the right place. Watson et al. [19] have proposed the term *u-commerce* (or *ubiquitous commerce*), which is defined as "*the use of ubiquitous networks to support personalized and uninterrupted communications and transactions between a firm and various stakeholders to provide a level of value over, above and beyond traditional commerce.*" A number of architectures and prototypes of u-commerce systems have been described in the literature [3]. The *Shopper's Eye* [2] proposed a location-aware service with wirelessly enabled portable terminals, e.g., PDAs and smart phones. As a shopper travels about, his or her personal terminal transmits messages, which include information about his or her location, shopping goals, preferences, and related purchase history. When this information is received, stores create a customized offer of goods and services. The Impulse project [18] is a PDA-based system whereby customers may add products to a list, indicating preferences such as warranty terms, merchant reputations, availability, time limits for the purchases and preferred price. When a potential customer enters shopping zones, individual stores, or shopping malls, his/her agent engages nearby merchants in a silent exchange seeking items on the list and opens negotiations on the terms of the sale, alerting the shopper if a deal has been agreed on. It is envisaged that the merchant gains valuable information about customers' purchasing behavior during the negotiation process.

That system presented in this paper is an application of our previous bio-inspired system [13]. 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 papers [12, 14], but did not describe any nature-inspired deployment policies in those works.

3 BASIC APPROACH

This section outlines our approach to federate smart objects according context in the real world.

3.1 Nature-inspired federation of smart objects

Smart objects are everyday objects that are equipped with hardware components such as wired or wireless interfaces for communication through standard protocols, e.g., TCP/IP, a CPU to process tasks, sensors/actuators to be conscious of the world in which they are situated and to control it at a given instant. There are two key ideas behind the middleware system resented in this paper. 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. For example, location-aware annotation services on exhibits designed for running on mobile terminals are often activated only when their users are close to the exhibits. At the same time, the contents of the annotation services themselves may be able to be used on stationary terminals. 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. They are defined as relocation relationships between two service-provider agents or between a service and virtual counterpart agents.

3.1.1 Virtual counterparts. To introduce the metaphors of gravitational and repulsive forces to context-aware services, we abstract away the underlying systems, including location-sensing systems.

Our middleware system has the following two kinds of agents, discussed below.

- 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.
- 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, individually for each 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 [15] 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.

3.1.2 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 use *repulsive* force between service-provider agents.

4 DESIGN AND IMPLEMENTATION

Our middleware system consists of two parts: (1) context information managers, (2) runtime systems for application-specific services as shown in Fig. 2. 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 systems 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.

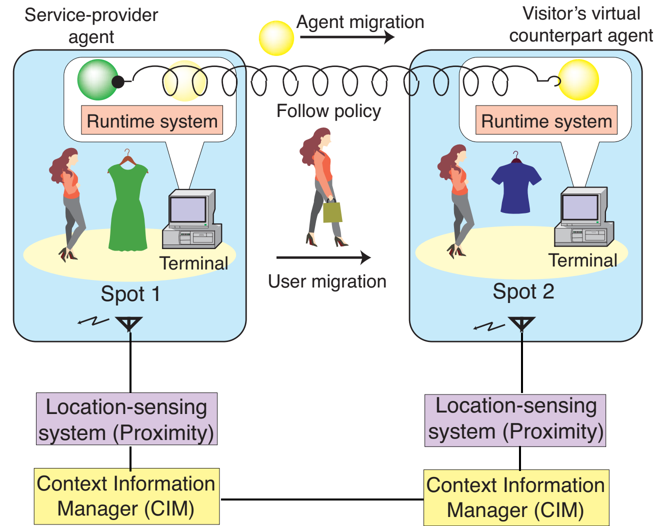

Figure 2: Architecture.

4.1 Context information manager

Each context information manager (CIM) manages one or more sensing systems to monitor context in the real world, 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 smart cities.

Therefore, this paper focuses on sensing location of smart objects and users although the manager itself can support a variety of context. The current implementation of CIM supports active 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.

4.2 Runtime system

Runtime systems migrate agents to other runtime systems running on different computers through TCP channels using mobile-agent technology [15].

4.2.1 Execution and deployment of services. Each runtime system is built on Java virtual machine (Java VM) version 1.7 or later, which conceals differences between the platform architectures of the source and destination computers (Fig. 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.

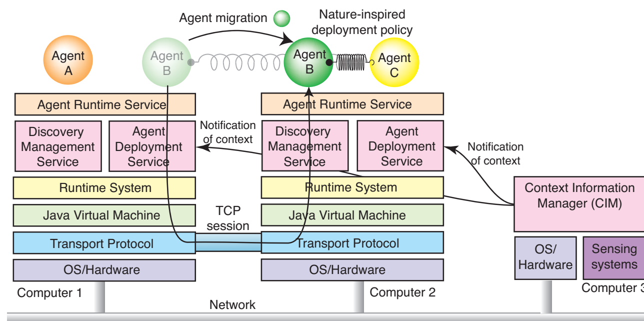

Figure 3: Runtime system

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.

4.2.2 Policy-based federation and deployment. Our nature-inspired 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 UDP multicasting, and these runtime systems then return their addresses and capabilities to the runtime system through a TCP channel.[1] The procedure involves four steps. 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. 3-a) 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. 3-b) 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. 4) 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.

4.3 Service

Each mobile agent is attached to at most one visitor and maintains that visitors's 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.[2] Each agent must be an instance of a subclass of the `Agent` class.

[1] We assumed that the agents comprising an application would initially be deployed at runtime systems within a localized space smaller than the domain of a sub-network.
[2] JavaBeans can easily be translated into agents in this platform.

```
class Agent extends MobileAgent implements Serializable {
  void go(URL url) throws NoSuchHostException { ... }
  void duplicate() throws IllegalAccessException { ... }
  setPolicy(ComponnetProfile cref,
    MigrationPolicy mpolicy) { ... }
  setTTL(int lifespan) { ... }
  void setAgentProfile(AgentProfile cpf) { ... }
  boolean isConformableHost(HostProfile hfs) { ... }
  void send(URL url, AgentID id, Message msg)
    throws NoSuchHostException, NoSuchAgentException, ... { ... }
  Object call(URL url, AgentID id,
    Message msg) throws NoSuchHostException,
      NoSuchAgentException, ... { ... }
  ....
}
```

Each agent can execute `go(URL url)` to move to the destination specified as a `url` by its current platform, and `duplicate()` creates a copy of the agent, including its code and instance variables. The `setTTL()` specifies the life span, called the time-to-live (TTL), of the agent. The lifespan decrements TTL over time. When the TTL of an agent reaches zero, the agent automatically removes itself.

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.

4.4 Current Status

A prototype implementation of this framework was constructed with Sun's Java Developer Kit, version 1.5 or later version. Although the current implementation was not constructed for performance, we evaluated the migration of a context-aware container based on connectors. When a container declares a *follow* or *shift* connector for a virtual-counterpart, the cost of migrating the former to the destination or the source of the latter after the latter has begun to migrate is 88 ms or 85 ms, where three computers over a TCP connection is 32 ms.[3] This experiment was done with three computers (Intel Core 2 Duo 2 GHz with MacOS X 10.6 and Java Development Kit ver.6) connected through a Fast Ethernet network. Migrating containers included the cost of opening a TCP-transmission, marshalling the agents, migrating them from their source computers to their destination computers, unmarshalling them, and verifying security.

5 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.

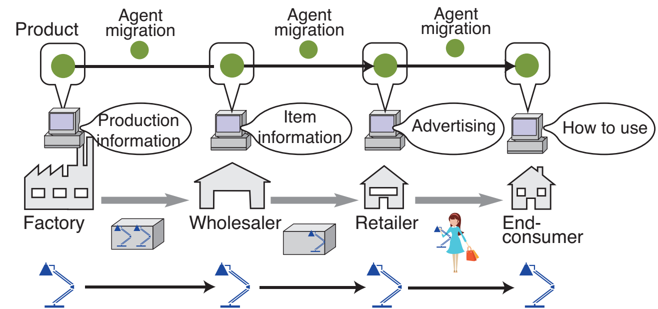

Figure 4: Forwarding agents to digital signage when user moves.

- **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. Figures 5 a) and b) have two images 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 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. Figures 5 a) and b) had two images maintained in the service that

[3]The size of each virtual-counterpart was about 8 KB in size.

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. Figure 5 c) has the active media for advice on assembly. The service also advises how it was to be used as shown in Fig. 5 d). When it was disposed of, the service presents its active media to give advice on disposal. Figure 5 e) has an image that illustrates 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.

Figure 5: Active media for appliance

6 CONCLUSION

This paper presented a context-aware service platform with a nature-inspired or self-organizing approach. The system enabled two individual agent to specify one of the deployment policies as relocation 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 in a store. 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.

REFERENCES

[1] O. Babaoglu and 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, July 2002.
[2] A. Fano: Shopper's eye: using location-based filtering for a shopping agent in the physical world, In Proceedings of International Conference on Autonomous Agents, pp. 416-421, ACM Press, 1998.
[3] H. Galanxhe-Janaqi and F. F-H. Nah: U-commerce: emerging trends and research issues, Industrial Management and Data Systems, Vol. 104, No. 9, pp.744-755, 2004.
[4] 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.
[5] 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, 2016.
[6] B. L. Brumitt, B. Meyers, J. Krumm, A. Kern, S. Shafer, EasyLiving: Technologies for Intelligent Environments, Proceedings of International Symposium on Handheld and Ubiquitous Computing, pp. 12-27, 2000.
[7] 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, 2016.
[8] G. Fortino, A. Guerrieri, W. Russo, C. Savaglio, Middlewares for Smart Objects and Smart Environments: Overview and Comparison, Internet of Things Based on Smart Objects pp.1-27, Springer, 2014.
[9] G. Fortino, M. Lackovic, W. Russo, 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, Springer, LNCS, Vol.8223, 2013.
[10] 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, 2002.
[11] T. Leppaanen, 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.
[12] I. Satoh, Context-aware Agents to Guide Visitors in Museums, in Proceedings of 8th International Conference Intelligent Virtual Agents (IVA'08), Lecture Notes in Artificial Intelligence (LNAI), vol.5208, pp.441-455, September 2008.
[13] 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, November 2008.
[14] 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.
[15] I. Satoh: Mobile Agents, Handbook of Ambient Intelligence and Smart Environments, pp.771-791, Springer 2010.
[16] A. Savidis and C. Stephanides, Dynamic environment-adapted mobile interfaces: the Voyager Toolkit, Proceedings of the 10th International Conference on Human-Computer Interaction (HCI International 2003), pp.489–493, 2003.
[17] P. Tandler: The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments, Journal of Systems and Software - Special issue: Ubiquitous computing archive Vol.69, No.3, pp.267-296, 2004.
[18] G. Tewari, J. Youll, and P. Maes: Personalized location-based brokering using an agent-based intermediary architecture, Decision Support Systems, Vol. 34, No. 2, 127-137, 2003.
[19] R. E. Watson, L. F. Pitt, P. Berthon, and G. M. Zinkhan: U-commerce: expanding the universe of marketing, Journal of the Academy of Marketing Science, Vol. 30, No. 4. 333-347, 2002.