Dynamic Federation of Partitioned Applications in Ubiquitous Computing Environments

Ichiro Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
E-mail: ichiro@nii.ac.jp

Abstract

A framework for the dynamic aggregation of ubiquitous computers is presented. The framework aggregates applications from more than one mobile component that can migrate from computer to computer during the execution of applications. Using location-tracking systems, the framework can spatially bind partitioned applications to users and other things and dynamically allocate and migrate a federation of partitioned applications on multiple computers according to the capabilities and locations of the computers and the positions of users. The framework also provides inter-component communications and component relocation semantics so that an application made up components can adapt its structure and functionality to changes in resource availability and the physical environment. A prototype implementation of the framework built on a Java-based mobile agent system and its applications are described.

1. Introduction

An important research issue in ubiquitous computing concerns the progress being made toward developing of an application environment that is able to deal with the mobility and interactions of both users and devices. Rapid advances in device technologies and falling costs are enabling a variety of computers to be linked through wired or wireless networks in modern offices and homes. Users are surrounded by hundreds of computers from desktop PCs to small computers embedded in artifacts, and by sensors that are able to acquire information from the physical world. However, these computers cannot always support applications other than those they were not initially designed for because their computational resources, such as processors, storage, and input and output devices, are limited as they were only optimized for their initial purposes. To accomplish goals beyond the capabilities of individual computers, a ubiquitous computing application should not only be able to be processed by a single computer but also by the interaction of a group of computers, called a federation. Moreover, such a group must be configurable in runtime because the requirements of users may change dynamically. This paper presents a framework that enables ubiquitous computers to be dynamically federated. The framework facilitates the construction of a virtual computer as a federation of partitioned applications around different computers. It also enables partitioned applications to be deployed at, and run on, heterogeneous computers that can provide the computational resources required by users and their associated context, such as locations and tasks.

In the remainder of this paper, we describe our design goals (Section 2), the design of our framework, called Hydra, and a prototype implementation (Section 3) and an application of the framework (Section 4). We briefly review related work (Section 5), provide a summary, and discuss some future issues (Section 6)

2. Architecture Overview

Our framework enables us to construct an application as a federation of ubiquitous computers connected through a network for overcoming the limitations of computational resources, such as input and output devices and restricted processors, in single ubiquitous computers (Figure 1).

Applications and partitioned applications must not be bound to ubiquitous computers, which have limited computational resources, for various applications, but they should run on computers that can satisfy their requirements. Mobile users may also constantly want to change the computers with which they interact. That is, applications should be able to move from computer to computer to follow users. Therefore, our framework should enable a federation of partitioned applications to migrate partially or entirely to suitable computers based on changes in users and their associated contexts, e.g., locations, current tasks, and the number of people.

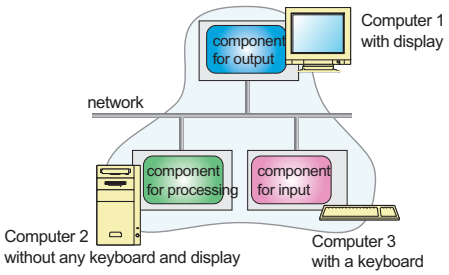

Figure 1. Federation for heterogeneous computers

The framework builds partitioned applications as mobile agent-based software components and enables these to move to other computers while the application is running. When an application is made up multiple components, the movement of one component may affect the others. It therefore provides three typical interactions: *publish/subscribe* for asynchronous event passing, remote method invocation, and stream-based communication to coordinate mobile components. It provides mechanisms for keeping these interactions even when some of the components move to other locations. Moreover, the deployment of components is often dependent on their applications. For example, two components are required to be at the same or nearby computers, when the first is a program that controls the keyboard and the second is a program that displays content on the screen. The framework therefore enables each component to specify explicitly a policy for component migration, called a *hook*. The current implementation provides two types of hooks, as we can see in Figures 2 and 3. The first means that a component follows another, and the second means that a component creates a copy of itself and makes the copy follow another component. Our framework can dynamically allocate a federation of partitioned applications at suitable computers by using these policies.

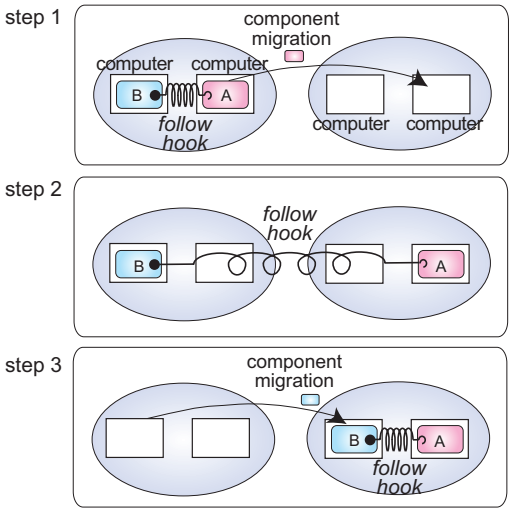

Figure 2. Follow policy in two components.

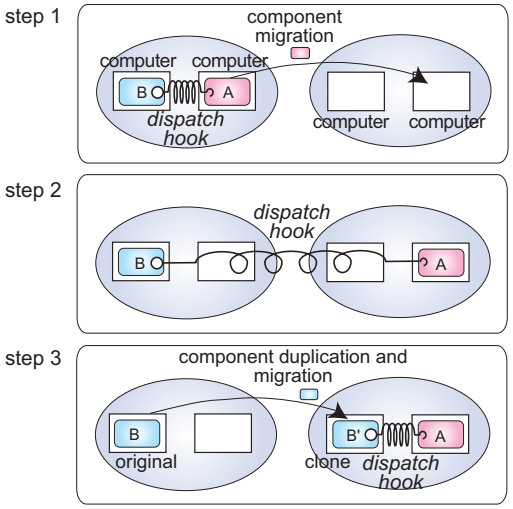

Figure 3. Dispatch policy in two components.

3. Design and Implementation

Our framework consists of two parts: components and component hosts.

3.1. Component

Automatically partitioning existing standalone applications across multiple computers is almost impossible. Instead, this framework relies on the concept of component-based application construction. That is, an application is loosely composed of software components, which may run on different computers. However, we have not assumed any application models in our framework unlike those in existing related work.

Each component in the current implementation of the framework 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.[1] Each is also equipped with its own identifier and the identifier of the federation that it belongs to, and it specifies the computational capability that its destination hosts must offer in composite capability/preference profiles (CC/PP) form to describe the capabilities of component hosts and

1 JavaBeans can easily be translated into components in the framework.

the requirements of components. The framework provides each component with built-in APIs to verify whether or not its destinations satisfy its requirement.[2] The APIs transform profiles into their corresponding LISP-like expressions and then evaluate them.

Each component can provide references to the other components of the application federation that it belongs to. Each reference allows a component to interact with the component that it specifies, even when the proceeding and following components are at different computers or when they move to other computers. The current implementation of the referencing provides three types of mobility-transparent interactions: publish/subscribe-based remote event passing, remote method invocation, and stream communication between computers. Moreover, each reference defines two migration policies for two components, *follow* hook and *dispatch* hook, as follows:

- When a component declares a *follow* hook for another component, if the following component moves, the hook instructs the proceeding one to migrate to the destination or to a proper host.
- When a component declares a *dispatch* hook for another component, if the following component moves, the hook instructs a copy of the proceeding one to migrate to the destination or a proper host.

where the above proper hosts correspond to computers whose capabilities can satisfy the requirements of the migrated component and that may be spatially near from the original destination when location sensing systems can locate computers. In the second hook, the original and clone components run independently. Our relocation constraint is similar to the dynamic layout of distributed applications in the FarGo system [3], but the latter aims to allow one or more components to control another, whereas the former aims to allow one component to describe its own migration, because our framework treats components as autonomous entities that travel under their control from computer to computer.[3]

3.2. Component Host

Each component host is a computer, and it provides a runtime system for executing and migrating components to other hosts. Each host establishes at most one TCP connection with each of its neighboring hosts and exchanges control messages, components, and inter-component communications with these through the connection.

Component Runtime Service Each runtime system is built on the Java virtual machine, which conceals the differences between the platform architecture of source and destination hosts, such as the operating system and hardware. Each runtime system governs all the components inside it and maintains the life-cycle state of each component. When the life-cycle state of a component changes, e.g., when it is created, terminates, or migrates to another host, the runtime system issues specific events to the component. This is because the component may have to acquire various resources or release them, such as files, windows, or sockets, that it had previously captured.

Component Migration Service Each component host can exchange components with another through a TCP channel with mobile agent technology. When a component is transferred over a network, a component host on the sending side marshals the code of the component and its state into a bit-stream and then transfers it to the destination. Another component host on the receiving side receives and unmarshals the bit-stream. The current implementation uses the standard JAR file format for passing components that can support digital signatures, allowing for authentication. It also uses Java's object serialization package for marshaling components. The package can save the content of instance variables in a component program but does not support the capturing of stack frames of threads. Instead, when a component is marshaled and unmarshaled, the component host propagates certain events to its components to instruct them to stop their active threads, and then it automatically stops and marshals them after a given period of time.

Migration-transparent Coordination Service The framework also provides three interactions: publish/subscribe for asynchronous event passing, remote method invocation, and stream-based communication. Each runtime system offers a remote method invocation (RMI) mechanism through TCP connection, which is independent of Java's RMI because Java's RMI lacks any reference updating mechanism in migrating components. It also maintains a database that stores pairs of identifiers for its connecting components and the network addresses of the current component host.

Relocation Policy Management Service Each component host periodically advertises its address around the other component hosts by means of UDP multicasting, and then these hosts return their addresses and capabilities, which are written in CC/PP forms, to the host through a TCP channel.[4] Next, we show how to relocate components. (1) When arriving at a component host, each component automatically

2 For space reasons, detailed information is left to other papers [5].

3 This difference is important, because FarGo policies may conflict if two components can declare different relocation policies for one single component, whereas our framework is free from any conflicts because each component can only declare a policy for its own relocation instead of other components.

4 We assume components that an application consists of initially are deployed at hosts within a localized space smaller than the domain of a sub-network for multicasting packets.

registers its policy with the host. (2) The host then sends a query message to neighboring hosts that it knows have discovered the component that in the policy or the proxy of the related component the visiting component is related to. (3-a) If a host has a component specified in the policy, it returns information about itself and neighboring hosts that it knows, e.g., network addresses and capabilities, within its current networked or spatial domain to the source host of the message. (3-b) If a host has the proxy of a component specified in the policy, it forwards the query message to the destination of the component. (4) When the specified component migrates to another location, the destination host sends information about itself and neighboring hosts that it knows to the host that sent the query message about the component. (5) Each component, or its clone, selects one host from the candidate destinations recommended by component hosts and migrates to the selected host because this framework treats every component as an autonomous entity. Moreover, when the capabilities of a candidate destination do not satisfy all the requirements of the component, the component itself should decide on the basis of its own configuration policy whether or not it will migrate itself to the destination and adapt itself to the destination's capabilities.

3.3. Component Programming

Each component was implemented as a collection of Java objects. Also, each component needed to be an instance of a subclass of the `MobileComponent` class. Here, we while explain some programming interfaces that characterized the framework.

```
class MobileComponent extends MobileAgent
  implements Serializable {
  void go(URL url)
    throws NoSuchHostException { ... }
  setPolicy(ComponnetProfile cref,
    MigrationPolicy mpolicy) { ... }
  ComponentHost[] getDestinationHosts(
    Domain dm) { ... }
  void setComponentProfile(
    ComponentProfile cpf) { ... }
  boolean isConformableHost(
    CCPPHostProfile hp) { ... }
  ComponentProfile getComponentProfile(
    ComponentRef ref) { ... }
  ....
}
```

Let us briefly explain some methods defined in the above class.

- A component executes the `go(URL url)` method to move to the destination host specified as the `url` by its runtime system.
- Each component can declare its own migration policy by invoking the `setPolicy()` method of the `Component` class while it is running as follows:

```
setPolicy(cref,
  new MigrationPolicy(Policy.FOLLOW));
setPolicy(cref,
  new MigrationPolicy(Policy.DISPATCH));
```

  The framework is open to the introduction of new policies as long as they are subclasses of `MigrationPolicy` for defining the migration policy.
- The `getComponentHosts()` returns a list of the component hosts that satisfy the requirement and are within a given domain specified as an instance of the class `Domain`, which can defines a spatial scope and network domain.
- Each component can specify a requirement that its destination hosts must satisfy by invoking the `setComponentProfile()` and can easily decide whether or not the capabilities of the component hosts specified as an instance of the `CCPPHostProfile` class satisfy its requirements by invoking the `isConformableHost()` method.

4. Initial Experience

This section outlines a typical mobile application developed with the framework. The application is a mobile editor and is composed of three partitioned components. The first, called *application logic*, manages and stores text data and should be executed on a host equipped with a powerful processor and a lot of memory. The second, called a *viewer*, displays text data on the screen of its current host and should be deployed at hosts equipped with large screens. The third is called a *controller* and forwards texts from the keyboard of its current host to the first component. They have the following relocation policies. The application logic and control components have *follow* hook policies for the viewer component to deploy themselves at the current host of the viewer component or nearby hosts. As we can see from Figure 4, we assume that the three components are initially stored in two hosts.

The system can track the movement of the user in a physical space through an RFID-tag technology and introduces a component, called a *user-counterpart*, that can automatically move to hosts near the current location of the user, even while the user is moving. That is, a user-counterpart is always at a host near the user. Because the viewer component had a *follow* hook policy to move the user-counterpart component, it moves to the host that has the user-counterpart or nearby hosts. When a user moves to another location, the components could be dynamically allocated at suitable hosts without losing any coordination between them as we show in Figure 4. Although the current implementation was not built for performance, we measured the group migration of the four components. The latency of migrating the three components was 240 msec after migrating the counterpart component, where the cost of

component migration between two hosts over a TCP connection was 40 msec.[5]

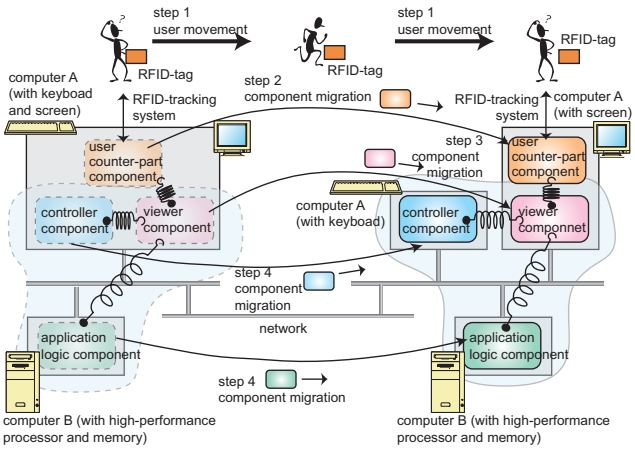

Figure 4. Initial allocation of components for editor-application.

5. Related Work

A research trend in pervasive computing is to aggregate computational resources attached to different computers. Several projects for the aggregation of computers in pervasive computing settings have been explored. For example, EasyLiving [1] provides middleware for dynamically aggregating networked-enabled input/output devices, such as keyboards and mice, even when they are used with to different computers. BEACH [6] is middleware for constructing collaborative applications through shared or distributed devices. Both approaches cannot dynamically deploy components around different computers. Aura [2] is an infrastructure for binding tasks associated with users and migrating applications from computer to computer as users move about. It focuses on providing contextual services for users rather than integrating multiple computers to support function and performance unattainable through a single computer. Gaia [4] is an infrastructure for allowing applications to be partitioned between different computers and moving from computer to computer under its centralized server instead of the applications themselves. Most existing approaches, including BEACH and Gaia, assume that applications are inherently designed based on the model-view-control (MVC) approach, but many modern applications are constructed based on more complex application models, e.g., design patterns, rather than the traditional MVC model. Moreover, these existing systems assume that centralized systems for managing computers exist so that they cannot support the requirement of each individual application. They are also not always scalable in a widespread building-wide or city-wide system. To solve these problems, the framework introduces the notion of relocation constraint, called the *hook* policy. This notion enables a federation of components to be organized among heterogeneous computers in a self-organized manner.

5 This experiment was done with five component hosts (Pentium III-1.2 GHz with Windows XP and JDK 1.4) connected through a Fast Ethernet network.

6. Concluding Remarks

A novel framework for dynamically aggregating distributed applications in ubiquitous computing environments was discussed. It is used to build an application from mobile agent-based components, which can explicitly have policies for their own deployment. It also supports most typical interactions between partitioned applications on different computers. It enables a federation of components to be dynamically structured in a self-organized manner and to move among heterogeneous computers that can provide the computational resources required by the components. We designed and implemented a prototype system for the framework and demonstrated its effectiveness in several practical applications.

References

[1] 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 (HUC'00), pp. 12-27, September, 2000.

[2] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, Project Aura: Towards Distraction-Free Pervasive Computing, IEEE Pervasive Computing, vol. 1, pp. 22-31, 2002.

[3] O. Holder, I. Ben-Shaul, and H. Gazit, System Support for Dynamic Layout of Distributed Applications, Proceedings of International Conference on Distributed Computing Systems (ICDCS'99), pp 403-411, IEEE Computer Society, 1999.

[4] M. Román, C. K. Hess, R. Cerqueira, A. Ranganat, R. H. Campbell, and K. Nahrstedt, Gaia: A Middleware Infrastructure to Enable Active Spaces, IEEE Pervasive Computing, vol. 1, pp. 74-82, 2002.

[5] I. Satoh, Physical Mobility and Logical Mobility in Ubiquitous Computing Environments, Proceedings of 6th International Conference on Mobile Agents (MA'2002), LNCS, vol. 2535, pp. 186-202, Springer, October 2002.

[6] P. Tandler Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices, Proceedings of UbiComp'2001, LNCS, vol. 2201, pp. 96-115, Springer, 2001.