Dynamic Deployment of Pervasive Services

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

Abstract— **This paper presents a self-organizing approach to developing and managing distributed software in pervasive computing environments. In such environments, people are surrounded by hundreds of mobile or embedded computers each of which may be used to support one or more user applications due to limitations in their individual computational capabilities. We need an approach to coordinating heterogeneous computers that acts as a virtual computer around a mobile and pervasive computing environment and supports various applications beyond the capabilities of single computers. This paper presents a framework for building and aggregating distributed applications from one or more mobile components that can be dynamically deployed at mobile or stationary computers during the execution of the application. Since the approach involves mobile-transparent communications between components and component relocation semantics, it enables a federation of components to adapt its structure and be deployed on multiple computers whose computational resources, such as input and output devices, can satisfy the requirement of the components in a self-organized manner. This paper also describes a prototype implementation of the approach and its application.**

I. Introduction

Our research on pervasive computing is concerned with our progress toward developing a pervasive computing service that is able to deal with the mobility and interactions of both users and devices. Advances in device technologies and falling costs are rapidly enabling a variety of computers that are networked through wired or wireless networks to be provided in modern offices and homes. Users are surrounded by hundreds of computers from desktop PCs to small computers embedded in artifacts, and by sensors able to acquire information from the physical world. However, these computers cannot always support services that they were not initially designed for, because their computational resources, such as processors, storage, and input and output devices, are limited having only been optimized for their initial purposes. To accomplish goals beyond the capabilities of individual computers, a pervasive computing service should be able to be processed not only by a single computer but also by interaction between a group of computers, called a *federation*. Moreover, such a group must be configurable in runtime because the goals and positions of users may change dynamically.

This paper presents a framework for the dynamic federation of pervasive computers, called Hydra. The goal of the framework is to facilitate the construction of a virtual computer formed by a networked set of pervasive computers. This may seem to be similar to the notion of grid computing in the sense that a virtual computer is organized from distributed computers, but grid computing is aimed at a very large-scale, generalized distributed computing system that can be scaled to Internet size environments with machines distributed across multiple organizations and administrative domains. Our framework is aimed at virtually connecting several devices that are actually attached to different computers in localized areas that will run interactive services to assist users. There have been several attempts at integrating the user interface devices of different computers on a network [2], [5], [9], [13], [25]. However, most existing attempts aim at enabling an application to directly redirect its application output events to output devices, which may be attached to another computer, and redirect input events from input devices, which may be attached to a different computer, through a network. Since there tend to be many naive events from input devices, and to output devices, these attempts have resulted in network traffic and latency. Some of these have been specialized for particular applications. Our framework, however, is aimed at supporting a variety of application partitions and enabling application partitions to exchange their application-level events.

Moreover, mobile users often want to constantly change the computers with which they interact. Consequently, pervasive services should be able to move from computer to computer to follow users. The requirements of the services also tend to vary and changed dynamically. The structure of a pervasive computing environment may also be changed frequently by adding or removing components and changing the network topology. Therefore, our framework enables a federation of partitioned applications to partially or entirely migrate to suitable computers according to changes in users and their associated contexts, e.g., the positions and numbers of people and computing devices.

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). We outline programs in the system, applications running on it (Section 4), and our current status (Section 5). We illustrate two applications of the framework (Section 6). We briefly review related work (Section 7), provide a summary, and discuss some future issues (Section 8)

II. Architecture Overview

Our framework enables us to construct an application as a federation of pervasive computers connected through a network. The framework also introduces two notions: distributed application partitioning and application mobility as

a unified approach to overcoming the limitations of computational resources, such as input and output devices and non-powerful processors, in single pervasive computers, as seen in Fig. 1. Since network latency between different computers through a wired or wireless network, including a power-line-line network, WiFi, and Bluetooth, may be non-negligible, the framework must only exchange essential information between partitioned applications rather than naive data redirected from input devices, and to output devices.

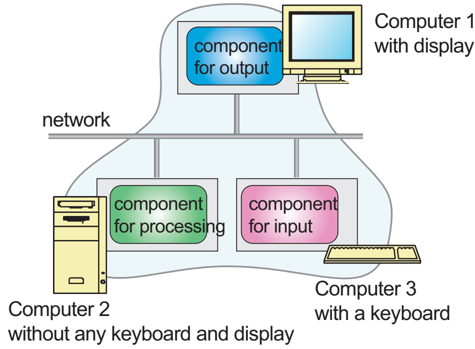

Fig. 1. Federation of heterogeneous computers.

A. Application Partitioning for Pervasive Computing

How to partition an application into multiple computers is a major decision in its design. A well known fact is that an application should be partitioned according to its functions. Most existing approaches to pervasive computing applications, e.g., Gaia [14] and BEACH [25], assume that applications are inherently designed based on particular application models, which are extensions of the model-view-control (MVC) model [10]. Therefore, they must be able to naturally divide their target applications into components corresponding to application logic, output, and input parts, before deploying these components at different computers. However, modern applications are often constructed based on more complex application models, e.g., design patterns [3], rather than the traditional MVC model. Pervasive computing may also need new or special design patterns as several researchers have discussed [11].

Therefore, our framework should not assume particular application models. However, most interactions between components in object-oriented applications within a computer can be covered by three primitives, i.e., event passing, method invocation, and stream communication. Therefore, the framework enables the three primitives to be available in partitioned applications on different computers. Achieving syntactic and (partial) semantic transparency for remote interactions requires the use of some proxy element that has the same interfaces as the remote components themselves.

B. Mobile Applications in Pervasive Computing Environments

Applications and partitioned applications must not be bound to pervasive computers, which have limited computational resources for various applications, but should run on computers that can satisfy their requirements, according to changes in users and their associated contexts, e.g., locations, current tasks, and the number of people. Therefore, the framework is used to build partitioned applications as mobile agent-based software components and enables these to move to other computers while the applications are running. The framework introduces such objects, called references, to possibly track moving targets and to interact with these through the three primitives. Moreover, the deployment of components is often dependent on their applications. That is, when an application is made up of multiple components, the movement of one may affect the others. 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 explicitly specify a policy for component migration, called a *hook*. The current implementation provides two types of hooks. The first enable a component to follow another component as we can see in Fig. 2, and the second enables a component to create a copy of itself and makes the copy follow another component as shown in Fig. 3.

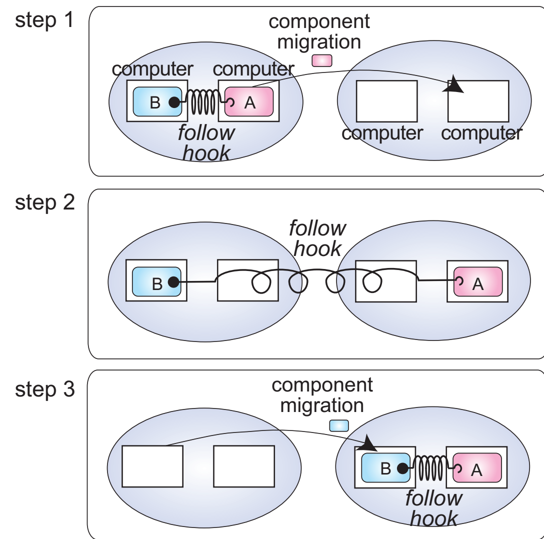

Fig. 2. Follow policy with two components.

The second policy assumes the copy of the component to be independent of its original, because the policy is used to deploy components at remote computers. Our framework can dynamically allocate a federation of partitioned applications at suitable computers by using these policies. Fig. 4 has an example of a group migration of three components. When component B has a *follow* policy for component A and component C has a *dispatch* policy for component A, if component A moves, component B moves to component A's destination host because the host satisfies component B's requirements. Also, a copy of component C moves to a nearby proper host that satisfies component C's requirements. The first policy is useful when relationships between components that

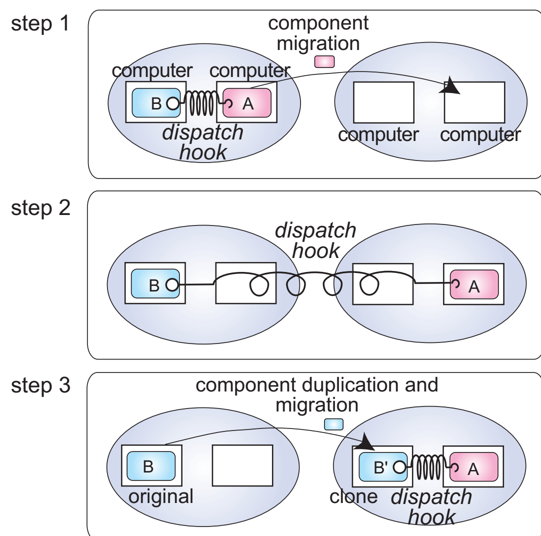

Fig. 3. Dispatch policy with two components.

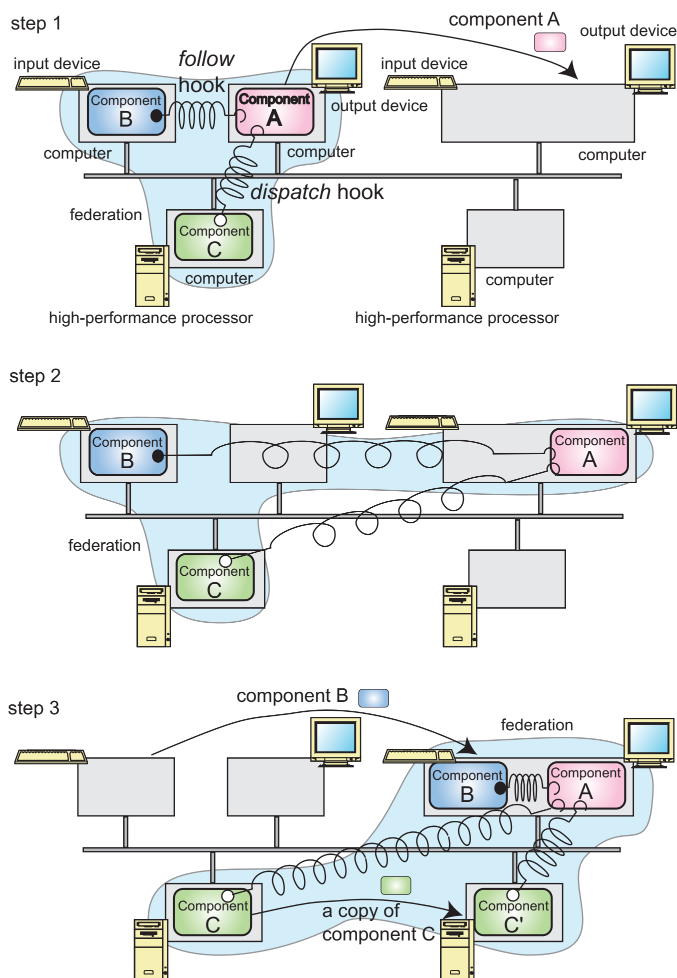

Fig. 4. Component migration with relocation policy

an application consists of should be retained and the second policy is useful when components are distributed at computers along the track of moving components. Several researchers have explored mechanisms for dynamically deploying components. Our mechanism enables each component to specify its deployment policy, whereas most existing mechanisms assume a centralized management system to control the deployment of components.

C. Location-aware Deployment of Mobile Applications

Since pervasive computers will be used in a variety of locations and situations, certain input and output modalities may be more appropriate in different circumstances. Some computers may be dynamically added to or removed from a localized space. For example, pen-based input operations may be a good idea when the user is standing in a public space or building but not when the user is driving a car. If there is a keyboard in a room, the user may not able to access this far from his/her current position. That is, user-interface devices near the users need to be selected and used. Our framework therefore dynamically allocates a federation of partitioned applications at suitable computers according to changes in the locations of users and the locations and capabilities of computers. The current implementation of our framework offers a location-aware infrastructure in which spatial regions can be determined within a meter, and that distinguishes one or more portions of a room or building through the use of RFID systems.[1] It determines the positions of objects by identifying the spatial regions that contain them. In general, such RFID systems consist of RF (radio frequency) sensors, often called *readers*, which detect the presence of small RF transmitters, often called *tags*. The current implementation also assumes that physical entities and places have been equipped with their own unique tags so that they are entities that can be automatically located.

[1]The framework itself is designed to be independent of any particular infrastructure for location and is accompanied by more than one locating system.

III. Design and Implementation

This framework consists of two parts: components and component hosts. The former denotes partitioned applications and the latter is middleware and enables components to run on a computer and migrate from computer to computer. The latter is independent of application-specific tasks because such tasks are confined within the former.

A. Component

Since it is almost impossible to automatically partition standalone applications across computers, this framework relies on the concept of constructing component-based applications [24]. That is, an application is loosely composed of software components that may run on different computers.

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 [16].[2] Each is also equipped with its own identifier and the identifier of the federation to which it belongs. Each also specifies the computational capability that its destination hosts must offer in CC/PP (composite capability/preference profiles) form [27], which describes the capabilities of the component host and the requirements of the components. The framework provides each component with built-in APIs to verify whether or not its destination satisfies its requirements.[3] The APIs transform the profiles into corresponding LISP-like expressions and then evaluate them.

Each component provides references to the other components of the application federation to which it belongs. Each reference enables the component to interact with the component that it specifies, even if the components are on different computers or move to other computers. The current implementation of 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, a *follow* hook and a *dispatch* hook.

- 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 same destination or an appropriate nearby host.
- When a component declares a *dispatch* hook for another component, if the following component moves, the hook instructs a clone of the proceeding one to migrate to the same destination or an appropriate nearby host.

The framework enables us to define other policies.

B. Component Host

Each component host is a computer and has 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. Fig. 5 outlines the basic structure of a runtime system. The current implementation can dynamically extend other component migration protocols by using the mobile software-based protocol mechanism [17]. Therefore, it can transmit the components through HTTP or SMTP by using tunneling techniques, because in almost all intranets, there is a firewall that prevents users from opening a direct socket connection to a node across administrative boundaries.

Component Runtime Service: Each runtime system is built on the Java virtual machine, which conceals the differences between the platform architectures of the 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

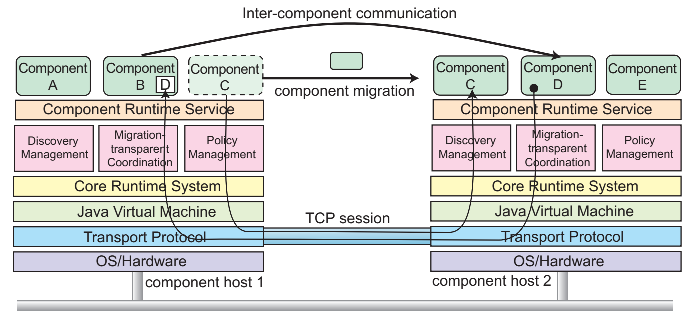

Fig. 5. Architecture for component host.

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, e.g., files, windows, or sockets, or release ones it had previously acquired.

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

Policy Management Service: The policy-based deployment of components is managed by each component host without a centralized management server. Each component host periodically advertises its address to the others through UDP multicasting, and then these hosts return their addresses and capabilities to the host through a TCP channel.[4] (1) When a component migrates to another component host, each component automatically registers its deployment policy to the destination host. (2) The destination host sends a query message to the source host of the visiting component. There are two possible scenarios: the visiting component has a policy for another component or it is specified in other component's policies. (3-a) In the former, since the source host knows the host running the target component specified in the policy of

[2]JavaBeans can easily be translated into components in the framework.

[3]More detailed information can be found to another paper [21], which was omitted have, due to space limitations.

[4]We assume that the components comprising an application would initially be deployed to hosts within a localized space smaller than the domain of a sub-network.

the visiting component, it asks the host to send the destination host information about itself and about neighboring hosts that it knows, e.g., network addresses and capabilities. If the target host has retained the proxy of a target component that has migrated to another location, it forwards the message to the destination of the component via the proxy. (3-b) In the second scenario, the source host multicasts a query message within current or neighboring sub-networks. If a host has a component whose policy specifies the visiting component, it sends the destination host information about itself and its neighboring hosts. (4) The destination host next instructs the visiting component or its clone to migrate to one of the candidate destinations recommended by the target 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 decides, on the basis of its own configuration policy, whether it will migrate itself to the destination and adapt itself to the destination's capabilities.

Migration-transparent Coordination Service: The framework provides three interactions: publish/subscribe for asynchronous event passing, remote method invocation, and stream-based communication.[5] Each runtime system offers a remote method invocation (RMI) mechanism through a TCP connection. It is independent of Java's RMI because Java's RMI lacks reference updating mechanisms for migrating components. Each runtime system can maintain a database that stores pairs of identifiers of its connected components and the network addresses of the current component host.

Each component host maintains a group of connected components, when one or more components migrate to other computers. The basic algorithm for migration-transparent communication is as follows. When a component migrates to another computer, it sends *suspend* messages to these hosts to block any new uplinks from them to the migrating component. It informs the current component host of the identifiers of components that may hold references to it. The component host then searches its database for the network addresses of component hosts with the components specified in the identifiers. After the component arrives at its destination, it sends an *arrival* message with the network address of the destination to the departure host via the destination host. When the departure host receives the arrival message, it sends `resumption` messages with the address of the destination to component hosts with components that may hold references to the moved component so that they can update their databases.

Most of the time, event dispatch passing or method invocation may complete each of their transactions instantly. However, components may experience extended delays due to network congestion or computational overload. Therefore, method invocation or event passing for moving components may occur or components that may hold references to a moving component may migrate to other places before the basic algorithm is completed. To solve this problem, a migrating component creates and leaves a proxy component at the departure host for the time the algorithm takes to finish. The proxy component receives uplinks from other hosts and forwards them to the moved component through the steps in Fig. 6, where components A and C have references to component B, which has a reference to component A and is moving to another location. Since no components have to be tracked for other components to communicate with them, components can leave proxy components along the trail under their control. Proxies are also programmable entities, like components, so they can modified based on application requirements.

[5]In the current implementation, remote event passing and stream-based communication have been implemented using our original RMI mechanism.

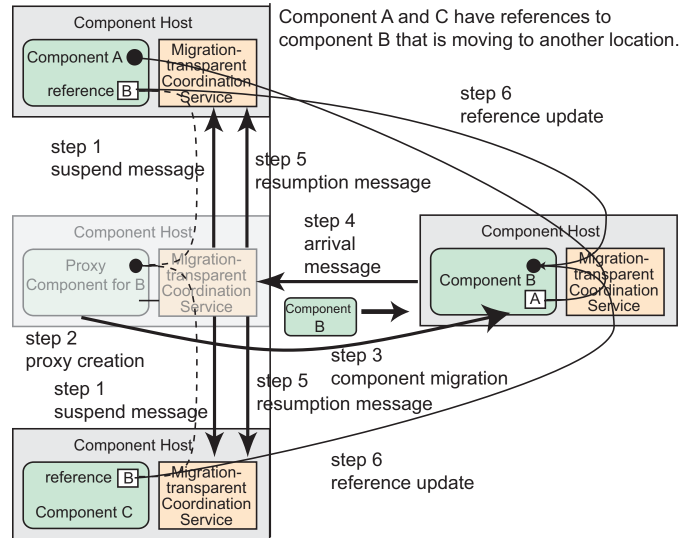

Fig. 6. Migration-transparency mechanism.

C. Location Information Service

In addition to components and component hosts, the framework provides location information servers (LISs) to deploy components at computers according to the positions of computers and physical entities, including people. Each LIS manages more than one sensor that detects the presence of RFID tags, and maintains up-to-date information on the identities of tags that are within its area of coverage.

Locating Sensors: Each LIS polls its sensors or receives events issued by them. To hide differences between underlying locating systems, each LIS maps low-level positional information from each of these in a symbolic model of location. An LIS represents an entity's location in symbolic terms of the unique identifier of the sensor that detects the entity's tag. Each sensor's coverage is called a *cell*, as in the models of location studied by several other researchers[12]. In the framework, multiple sensors do not have to be neatly distributed in a space, such as rooms or buildings, to completely it; instead, their coverage can overlap.

Component Discovery: Each LIS is responsible for discovering components bound to tags within its cells. It maintains

a database where it stores information about each of the component hosts and each of the components attached to a tagged entity or place. When an LIS detects a new tag in a cell, the LIS multicasts a query that contains the identity of the new tag and its own network address to all the component hosts in its current sub-network.[6] It then waits for reply messages from the component hosts. Here, there are two possible scenarios: it may be attached to a component host or the tag may be attached to a person, place, or thing other than a component host.

- In the first, the newly arriving component host sends its network address and device profile to the LIS; the profile describes the capabilities of the component host, e.g., its input devices and screen size. After receiving this reply, the LIS stores the profile in its database and forwards the profile to all component hosts within the cell.
- In the second case, some component hosts that have components tied to the new tag send their network addresses and the requirements of acceptable components to the LIS; the requirements for each component specify the capabilities of the component hosts that the component can visit and perform its services at.

If the LIS has no reply messages from the component hosts, it multicasts a query message to the other LISs. Figure 7 has a sequence for migrating a component to a proper host when an LIS detects the presence of a new tag. When an LIS detects the movement of a tag attached to a person or a thing, to a cell, it searches its database for component hosts that are present in the current cell of the tag. It then selects candidate destinations from the set of component hosts within the cell according to their respective capabilities and recommends candidate destinations for the components that the tag is attached to.

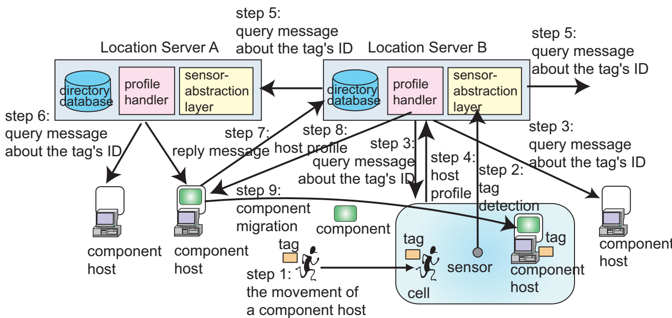

Fig. 7. Component discovery and deployment.

IV. Component Programming

As previously discussed, each component was implemented as a collection of Java objects and needed to be an instance of a subclass of the `MComponent` class. Here, we will explain some programming interfaces, which characterize the framework.

```
class MComponent extends MobileAgent
  implements Serializable {
  void go(URL url)
    throws NoSuchHostException { ... }
  void duplicate()
    throws IllegalAccessException { ... }
  void setGroupIdentifier(
    GroupIdentifier gid) { ... }
  GroupIdentifier getGroupIdentifier() { ... }
  void setComponentProfile(
    ComponentProfile cpf) { ... }
  boolean isConformableHost(
    HostProfile hfs) { ... }
  ComponentRef[] getGroupComponents() { ... }
  ComponentRef[] getComponents(
    Object cif) {..}
  ComponentProfile getComponentProfile(
    ComponentRef ref) { ... }
  setPolicy(ComponnetProfile cref,
  MigrationPolicy mpolicy) { ... }
  ....
}
```

Let us explain some methods defined in the `Component` class.

- A component executes `go(URL url)` to move to the destination host specified as `url` by its runtime system. `duplicate()` creates a copy of the component, including its code and instance variables.
- The `setGroupIdentifier()` ties the component to the identity of the federation specified as `gid` and the `getGroupComponents()` returns a list of components that belongs to the same federation. When `getComponents()` is invoked with an object corresponding to a component interface, it returns a list of components that can satisfy the interface in local and remote component hosts.
- Each component can specify a requirement that its destination hosts must satisfy by invoking the `setComponentProfile()`, with the requirement specified as `cpf`, where the requirement is defined in CC/PP form. The class has a service method named `isConformableHost()`, which the component uses to decide whether or not the capabilities of the component hosts specified as an instance of the `HostProfile` class satisfy the requirements of the component.

Each component can have more than one listener object that implements a specific listener interface to hook certain events issued before or after changes in its life-cycle state.

A. Component Migration Programming

Each component can declare its own migration policy by invoking the `setPolicy()` of the `Component` class while it is running.

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

For example, the upper command in the above code fragment means that when a component specified as `cref` moves to another computer, the component that executes the command

[6]Note that the coverage area of an RFID reader is mostly contained within the reachable domain of multicasting communications.

migrates to the same computer or nearby computers in the current cell that the computer resides at. The framework is open to the introduction of new policies as long as they are subclasses of `MigrationPolicy` that defines the migration policy.

B. Component Coordination Programming

Component references are responsible for tracking possible moving targets and for invoking the targets' methods. They are defined as the `ComponentRef` class and provide the following three primitives for remote interactions. Our early experience already proved that these primitives could cover most types of interactions presented in the literature on design patterns.

a) Remote Method Invocation:: The framework provides APIs for invoking the methods of other components on local or different computers with copies of arguments. Our programming interface for method invocation is similar to CORBA's dynamic invocation interface and does not have to statically define stub or skeleton interfaces through a precompiler approach, because pervasive computing environments are dynamic.

```
Message msg = new Message("print");
msg.setArg("hello world");
Object result = cref.invoke(msg);
```

The above code fragment is to invoke a method the component specifies as `cref` reference.

b) Publish/Subscribe-based Event Passing:: Modern GUI applications often rely on publish/subscribe approaches that provide subscribers with the ability to express their interest in an event so that they can be notified afterwards of any event notified by a publisher. The approach is useful in minimizing the number of events passed to remote computers. This framework provides a generic remote publish/subscribe approach using Java's dynamic proxy mechanism, which is a new feature of the Java 2 Platform since version 1.3.[7] Fig. 8 shows how events are passed, published by a component on a remote component host, to the corresponding event-listener object in another component on another remote component host.

```
SampleListener sl = new SampleListenerImpl();
cref.addListener(sl, "SampleListener");
```

The above code fragment registers the listener object specified as `sl`, which is an implementation of the `SampleListener` interface. The `addListener()` method dynamically creates a proxy object on a remote component host that has a remote component specified as `cref`. The proxy is an implementation of the `SampleListener` interface and automatically forwards events that are specified in the interface to the listener object on the local host.

c) Stream Communication: The notion of a stream is highly abstracted representing a connection to a communication channel. The framework enables two components on different hosts to establish a reliable channel through a TCP connection managed by the hosts.[8]

[7]As the dynamic creation mechanism is beyond the scope of this paper, we have omitted it here.

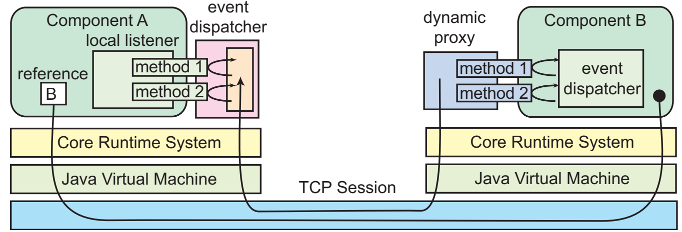

Fig. 8. Dynamic creation of proxy object for publish/subscribe

V. CURRENT STATUS

A prototype implementation of this framework was built with Sun's Java Developer Kit version 1.4.[9] It uses the Mobilespaces mobile agent system to provide mobile components and supports five commercial locating systems: RF Code's Spider, Aeroscout's WiFi-tag, Alien Technology's 950 MHz RFID-tag, Philips I-Code, and Hitachi's μ-chip systems. The first and seconds system provide active RFID-tags and the others provide passive RFID-tags. The Spider system's sensor can detect tags within a range of 1 to 20 meters. The Aeroscout's WiFi-tag can locate the position of tagged objects. Alien Technology's 915MHz RFID system's sensor can read tags within 3 meters and the other systems within within a few centimeters. The other systems can read tags within 0.3 meters.

Although the current implementation was not constructed for performance, we evaluated the group migration of three components (Fig. 4). The cost of migrating the three components was 180 msec, where the cost of migrating components between two hosts over a TCP connection was 40 msec and the cost of duplicating components in a host was less than 5 msec.[10] 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. The cost of migrating components included that of opening TCP-transmission, marshaling the components, compressing them in a zip-form, migrating them from their source hosts to their destination hosts, uncompressing them, unmarshaling them, and verifying security. We believe that this latency is acceptable for a location-aware system used in rooms or buildings.

The current implementation can encrypt components to be encrypted before migrating them over a network and then decrypt them after they arrive. Moreover, since each component is just a programmable entity, it can explicitly encrypt its particular fields and migrate itself with these fields and its own cryptographic procedure. The Java virtual

[8]Since our channel relies on TCP, it can guarantee exactly-once communication semantics across the migration of components.

[9]The functionalities of the framework except for subscribe/publish-based remote event passing can be implemented on Java Developer Kit version 1.1 or later versions, including Personal Java.

[10]Each of the three components was about 8 KB.

machine can explicitly restrict components to only access specified resources to protect hosts from malicious components. Although the current implementation cannot protect components from malicious hosts, the runtime system supports some authentication mechanisms for agent migration provided in mobile agent technology so that each component host can only send agents to and only receive from trusted hosts.

VI. Initial Experience

This section presents two examples that illustrate how the framework works.

A. Follow-Me Applications

The first example is a typical mobile application developed with the framework to illustrate how the this framework works. The application is a mobile editor. Since it is based on model-view-control (MVC) pattern [10], it is composed of three partitioned components corresponding to the model, view, and control parts in the pattern. The first, called application logic, manages and stores text data and should be executed on a host equipped with a powerful processor and a vast amounts of memory. The second, called viewer, displays text data on the screen of its current host and should be deployed at hosts equipped with large screens. The third, called controller, forwards texts from the keyboard of its current host to the first component.

These components interact with one another through the migration-transparent interaction primitives of this framework. The first component is a listener object for the third component that receives keyboard input. The second component is a listener object for the first component that updates its content according to changes in the stored first component. When it displays large amount of the first component's content, it establishes stream communication from the first component to itself to receive its content sequentially. They also have relocation policies as follows: the application logic and controller components have *follow* hook policies for viewer components to deploy themselves at the current host or nearby hosts.

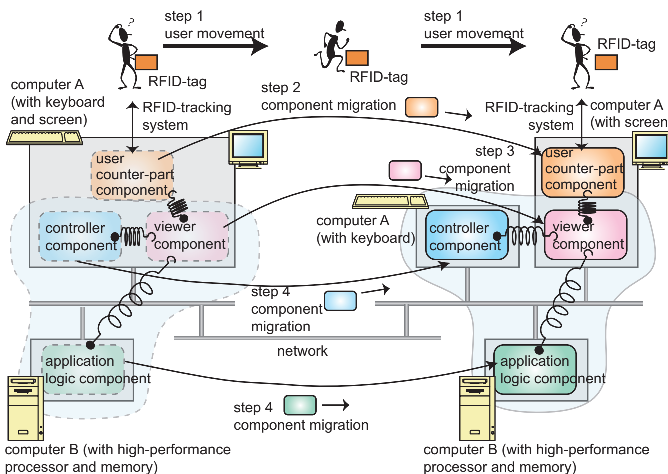

Fig. 9. Dynamic deployment of components for editor application.

As we can see from Fig. 9, we assumed that the three components would initially be stored in two hosts, where the first host had a keyboard and a large screen and the second one had a powerful processor and abundant memory. We developed a middleware for location-aware services in a previous paper [21]. The middleware could track the movement of the user in a physical space through RFID-tag technology.[11] Using the middleware and an RFID-tag system, we introduced a component called user-counterpart that could automatically move to hosts near the current location of the user, even while the user was moving. That is, a user-counterpart was always at a host near the user. Since the viewer component had a *follow* hook policy to move the user-counterpart component, it moved to a host that had a user-counterpart or nearby hosts. When a user moved to another location, the components could be dynamically allocated at suitable hosts without losing any coordination between them as we showed in Fig. 9.

B. Component Diffusion in Sensor Networks

The second example involves the speculative deploying of components according to changes in the physical world. This provides a mechanism that dynamically and speculatively deploys components at sensor nodes when there are environmental changes. This system assumes that the sensor field is a two-dimensional surface composed of sensor nodes and it monitors environmental changes, such as motion in objects and variations in temperature. It is a well known fact that after a sensor node detects environmental changes in its area of coverage, some of its geographically neighboring nodes tend to detect similar changes after a short time. Diffusion occurs as follows. When a component on a sensor node finds changes in its environment, the component duplicates itself and deploys the copy at neighboring nodes as long as the nodes have the same kinds of components (Fig. 10). Each component is associated with a resource limit that functions as a generalized Time-To-Live (TTL) field. Although a node can monitor changes in interesting environments, it sets the TTLs of its components as their own initial value. It otherwise decrements TTLs as the passage of time. When the TTL of a component reaches zero, the component automatically removes itself. This example is still in the early stages of experimentation but we have developed a mobile agent-based middleware for sensor networks [26] and plan to extend this framework to the middleware. Some readers may think that this mechanism is similar to dynamic management approaches for sensor networks [1], [8]. However, the existing approaches have only been optimized for only sensor networks and do not enable all deployable software to configure their deployment policies unlike our framework.

VII. Related Work

There have been several projects that have aggregated multiple-user-interface devices attached to different computers. For example, Microsoft's EasyLiving project [2] provided

[11]FID-based-tacking systems have been described in previous papers [21].

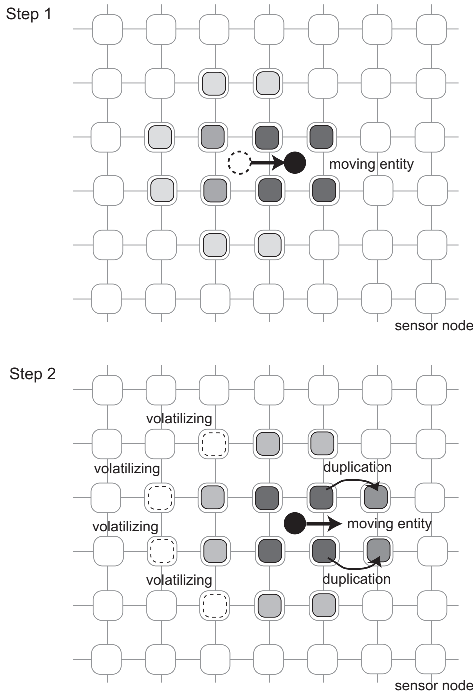

Fig. 10. Component diffusion with moving entity

a middleware, called InConcert, to dynamically aggregate networked-enabled input/output devices, such as keyboards and mice, even when they belonged to different computers. The middleware transmitted input and output across networks as necessary using devices that users indicated could be used. This is similar to the motivation behind our framework, but it managed these devices in a centralized manner and could not dynamically migrate software from computer to computer. The i-Land project by GMD-IPSI and Darmstadt University provided a component-based software infrastructure, called BEACH [25], which could provide support to construct collaborative applications through shared or distributed devices. BEACH, however, was aimed at synchronously sharing information between multiple users, and BEACH applications were statically bound to computers, unlike our framework.

Several researchers have introduced software mobility as a technology that enables pervasive computers to support various services, which they may not have initially been designed for. The Aura project [4] by CMU provides an infrastructure for binding tasks associated with users, and migrating applications from computer to computer as users move about, like our framework does. Although Aura shares several common design goals with our framework, it focuses on providing contextual services to users rather than integrating multiple computers to support functions and performance unattainable through a single computer. One.world [6] by the University of Washington and our MobileSpaces [16] provided an infrastructure for pervasive computing environments and mobile containers. This was called an *environment* by one.world and a *hierarchical mobile agent* by MobileSpaces, where each container not only provided the structure and controls for computational services but also acted as a carrier for the other containers. The two approaches assumed that each application would be executed within a single computer, instead of different computers.

The type of application partitioning we seek is similar to the one proposed by the Gaia project [14] of the University of Illinois at Urbana-Champaign. Like our framework, Gaia allows applications to be partitioned between different computers and move from computer to computer [15]. As discussed in Section II, Gaia assumes that applications will be constructed based on a design pattern, called MPACC, which is is an extension of the MVC pattern [10], whereas our framework supports a variety of interactions between partitioned applications so that we do not have to assume particular application models. Pervasive computing may also need new or special design patterns as several researchers have discussed [11]. Therefore, we have not assumed particular application models in our framework. Instead, it incorporates the notion of relocation constraint, called the *hook* policy. This notion enables a federation of partitioned applications to be organized among heterogeneous computers in a self-organized manner, unlike Gaia's applications.

The notion of our relocation policy may be similar to the dynamic layout of distributed applications in the FarGo system [7], but the former aims at allowing one component to describe its own migration, whereas the latter was aimed at allowing one or more components to control a single component. This is because our framework treats components as autonomous entities that travel from computer to computer under their own control. This difference is important, because FarGo policies may conflict if two components can declare different relocation policies for one single component. However, our framework is free of conflicts because each component can only declare a policy regarding its own relocation instead of those for other components. There have been numerous mobile agent systems in addition to the systems discussed in this section. They introduced mobile agents as independent computing entities that can travel between computers independently of the location of other agents. They, therefore, lack any mechanism for coordinating the relocation of one or more agents, running on local or remote computers.

We described an infrastructure for location-aware mobile agents in a previous paper [21]. Like the framework presented in this paper, that infrastructure provided RFID-tagged entities, including people and things, with application-level software to support and annotate them. However, since it could not partition an application into one or more components, it needed to deploy and run applications on single instead of multiple computers. We presented an earlier version of the framework presented in this paper in a recent short paper [22]. The previous framework aimed at building an application as a federation of one or more mobile components, but lacked migration-transparent coordination mechanisms and dynamic relocation policies supported by the current framework.

VIII. CONCLUSION

This paper discussed a novel framework for dynamically aggregating distributed applications in pervasive computing environments. It was used to build an application from mobile agent-based components, which could explicitly have policies to deploy themselves. It also supported most typical interactions between partitioned applications on different computers. It enabled a federation of components to be dynamically structured in a self-organized manner and move among heterogeneous computers that could provide the computational resources required by the components. We believe that the framework provides a general and practical infrastructure for building distributed and mobile applications. We designed and implemented a prototype system for the framework and demonstrated its effectiveness in several practical applications.

To conclude, we would like to identify further issues that need to be resolved. We aimed at presenting two deployment policies, but there are other and useful policies that may be possible. We are interesting at developing such deployment policies. The current policies treat a copy of the component to be running independently of the original, but we plan on providing a mechanism to enable a component and its clone to share updating. Although the examples presented in this paper were designed for single persons, we plan to implement multiuser applications, e.g., CSCW and workflow management systems. Security is essential in mobile applications and the current implementation of the system relies on Java's security manager. However, we plan to design a security mechanism that is more suited to distributed applications. We are also interested in security mechanisms that enable interactions between humans and components. We developed an approach to test context-aware applications on mobile computers [20], but need to develop a methodology for testing distributed applications that are based on this new framework. We also proposed a specification language for the itinerary of mobile software [23]. The language enabled us to define more flexible and varied policies to deploy components.

REFERENCES

[1] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T.W.D. Kim, B. Zhou, and E. G. Sirer, On the Need for System-Level Support for Ad hoc and Sensor Networks, Operating Systems Review, ACM, vol. 36, no.2, pp.1-5, April 2002.

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

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison-Wesley, 1995.

[4] 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.

[5] K. J. Goldman, B. Swaminathan, T. P. McCartney, M. D. Anderson, R. Sethuraman The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications, IEEE Transactions on Software Engineering, vol. 21, no. 9, pp.735-746, September 1995.

[6] R. Grimm, el. al., System support for pervasive applications, http://www.cs.nyu.edu/rgrimm//one.world.pdf

[7] 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 Soceity, 1999.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin, Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks, Proceedings of Conference on Mobile Computing and Networking (Mobicom'2000), pp.56-67, August, 2000.

[9] B. Johanson, G. Hutchins, T. Winograd, PointRight: Experience with Flexible Input Redirection in Interactive Workspaces, Proceedings of User Interface Software and Technology (UIST'02), ACM Press, 2002.

[10] G. E. Krasner and S. T. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigma in Smalltalk-80, Journal of Object Oriented Programming, vol.1 No.3, pp. 26-49, 1988.

[11] J. A. Landay and G. Borriello, Design Patterns for Ubiqutious Computing, Computer, vol. 36, no. 8, pp. 93-95, IEEE Computer Society, August 2003.

[12] Leonhardt U, Magee J. Towards a General Location Service for Mobile Environments. Proceedings of IEEE Workshop on Services in Distributed and Networked Environments, pp. 43-50, IEEE Computer Society, 1999.

[13] B.A. Myers: Using Hand-Held Devices and PCs Together, Communications of the ACM, vol. 44, no. 11, pp. 34-41, 2001.

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

[15] M. Román, H. Ho, R. H. Campbell, Application Mobility in Active Spaces, Proceedings of International Conference on Mobile and Ubiquitous Multimedia, 2002.

[16] I. Satoh, MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2000), pp.161-168, April 2000.

[17] I. Satoh, Adaptive Protocols for Agent Migration, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2001), pp.711-714, IEEE Computer Society, April, 2001.

[18] I. Satoh, MobiDoc: A Mobile Agent-based Framework for Compound Documents, Informatica, vol.25, no.4, pp.493-500, December 2001.

[19] I. Satoh, Building Reusable Mobile Agents for Network Management, IEEE Transactions on Systems, Man and Cybernetics, vol.33, no. 3, part-C, pp.350-357, August 2003.

[20] I. Satoh, A Testing Framework for Mobile Computing Software, IEEE Transactions on Software Engineering, vol. 29, no. 12, pp.1112-1121, December 2003.

[21] I. Satoh, Linking Physical Worlds to Logical Worlds with Mobile Agents, Proceedings of IEEE International Conference on Mobile Data Management (MDM'04), pp. 332-343, IEEE Computer Society, January 2004.

[22] I. Satoh, Dynamic Federation of Partitioned Applications in Ubiquitous Computing Environments, Proceedings of 2nd International Conference on Pervasive Computing and Communications (PerCom'2004), pp.356-360, IEEE Computer Society, March 2004.

[23] I. Satoh, Selection of Mobile Agents, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2004), pp.484-493, IEEE Computer Society, March 2004.

[24] C. Szyperski, D. Gruntz, and S. Murer, Component Software (2nd), Addison-Wesley, 2003.

[25] 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.

[26] Umezawa T, Satoh I, Anzai Y. A Mobile Agent-based Framework for Configurable Sensor Networks. Proceedings of International Workshop on Mobile Agents for Telecommunication Applications (MATA'2002); Lecture Notes in Computer Science 2002; Springer; Vol. 2521: 128-140.

[27] World Wide Web Consortium (W3C), Composite Capability/Preference Profiles (CC/PP), http://www.w3.org/TR /NOTE-CCPP, 1999.