Chapter 10

A Component-based Software Architecture for Pervasive Computing

Nalini Moti Belaramani, Yuk Chow, Vivien Wai-Man Kwan, Cho-Li Wang, Francis C.M. Lau
{moti, ychow, vjwmkwan, clwang, fcmlau}@csis.hku.hk
Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong

ABSTRACT

In recent years, we have observed an increasing prevalence of small devices equipped with wireless network connection. Computing is no longer a discrete activity bound to the desktop. Current software targeted for these devices however is limited by the devices' computing power and small memory, which in turn restricts the functionality and complexity of the applications that the device can provide. As a result, the current software system is not flexible and adaptive enough to satisfy the high demand due to increased user population and sophistication in pervasive environments. To attack the problem, we propose a component-based software architecture that supports dynamic mobile code binding and loading. An application is made up of several small and mobile code components – called facets. Whenever a specific function is needed during the execution of an application, a facet that implements that function is selected and brought in from the network, executed and then discarded. As a result, small devices are able to run applications of any sophistication. Moreover, the selection of facets depends on the computing resources in the device, network conditions, and various environmental contexts. This makes our system more flexible and adaptive, and capable to enhance user mobility in a pervasive computing environment.

INTRODUCTION

Computing has been an ever-changing paradigm since the beginning of its creation. With the millennium, there is an advent of a new computing environment. Computing is no longer limited to a “computer” per se. You see more and more different types of devices, such as personal digital assistants (PDAs) and mobile phones, taking advantage of wireless networks to connect to the Internet to provide services to the user.

A wealth of effort is going into the development of intelligent appliances and information appliances. More people are using mobile devices to access information or perhaps even just to communicate with each other. There is, no doubt, a trend towards more and more networked small devices with wireless access present in living and working spaces. The future, thus, will see great emphasis on pervasive computing (Neff, 2001 ; Pervasive Computing, 1999).

Pervasive computing can be summarized by 3 A’s – having access to computing and information Anywhere, Anytime and from Any device. The pervasive computing environment can be characterized by:

- Heterogeneity: Computing will be carried out on a wide spectrum of client devices, each with different configurations and functionalities.
- Prevalence of “small” devices: Many devices will be small, not only in size but also in computing power, memory size, etc.
- Limited network capabilities: Most of the devices would have some form of connection. However, even with the new networking standards such as GPRS, Bluetooth, 802.11x, etc., the bandwidth is still relatively limited compared to wired network technologies. Besides, the connections are usually unstable.
- High mobility: Users can carry devices from one place to another without stopping the services.
- User-oriented: Services would be tailored for the user rather than a specific device, or specific location.
- Highly dynamic environment: An environment in which users and devices keep moving in and out of a volatile network.

The whole environment can be seen as a huge ad-hoc distributed system, with a multitude of small devices moving from one place to another and cooperating with each other. With this new environment, new approaches have to be used to build applications. Current approaches to building distributed applications have been found to be flawed in a pervasive environment (Grimm *et al*., 2001). The pervasive computing environment poses new requirements on the infrastructure. These requirements are:

- Adaptation to diversity: The infrastructure should provide the ability for applications to adapt their functionality according to the device requirements, networks, etc.
- Increasing interaction with peers: Many of these devices will form ad-hoc networks among themselves in order to exchange information and to co-ordinate in order to provide services to the user.
- Flexible computation model: In a pervasive computing environment, there are various ways of accessing different types of data according to different users' needs. A combination of code and data mobility should thus be enabled to construct a flexible computation model.

Our project aims to build a user-oriented infrastructure designed specially with the needs of the future in mind. Our goals include:

- Client-dependent adaptability: Such a changing environment necessitates applications to be dynamic – to dynamically change according to user’s device configuration.
- User and device mobility: Users should be able to continue their work independent of their location or the device they are using.
- Peer-to-peer co-operative computing: With increased interaction between peer users, direct communication links within user peer groups can be established to support computations without any central control.

To achieve the above goals, a combination of software development techniques and infrastructural entities have to be used. The traditional software architecture for application development has to be changed. Instead of being huge monolithic chunks, software should be made up of smaller components. In addition, the network should have some intelligence to enable adaptability and provide for user-oriented services. Also, mobile code systems have to be incorporated in the software architecture, which utilize a combination of data and code mobility in order to achieve a flexible computation model.

SYSTEM OVERVIEW

Our system (http://www.csis.hku.hk/~clwang/projects/sparkle.html) utilizes a combination of software techniques and infrastructural entities to achieve the above goals. An overview of the proposed infrastructure is shown in Figure 1.

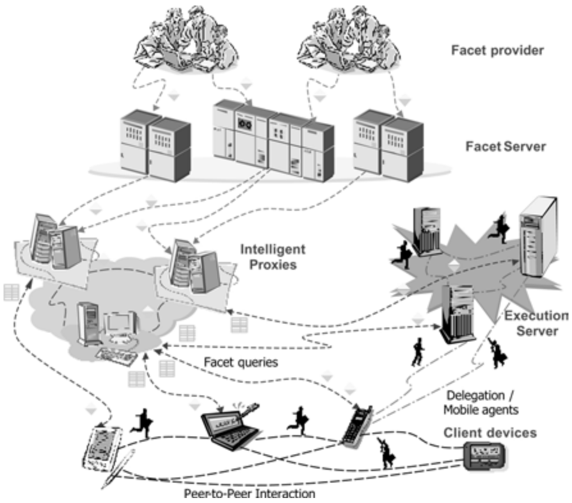

Figure 1: Overview of the overall system

Traditionally, applications are built as monolithic blocks. The problem is that they become too big to fit into small devices which have limited resources. Thus, a device's functionality is restricted by its configuration. Furthermore, such an approach does not allow for adaptability. Developers have to write programs tailored for each of their targeted client devices.

Some existing solutions (http://www.sun.com/software/sunone/; http://www.microsoft.com/net/) have adopted the web-services approach. Software is hosted on a server, and client devices access these services through the Internet. This requires transfer of data to the server to carry out computing. It falls short in cases in which the data should not be moved or is too large to be transferred over a slow network. This model requires a somewhat stable Internet connection which may not be always possible in a dynamic environment. Also, this model does not allow direct peer-to-peer communications.

Our approach is to use a dynamic component composition. Applications are built from small components. These components are downloaded when needed from the network at run-time and then cached for future use or thrown away after use. Thus, applications are dynamically composed at run-time from components. The advantage is that since every component is small, and can be thrown away, functionality in a device is not restricted by its limitations in configuration. Also, since the components are brought in at run-time, it allows applications to dynamically adapt to the client device. If there are two components of the same functionality, the component which is more suitable for the client device is brought in. Moreover, since functionality is brought in at run-time, it is not restricted to a particular device. If the user decides to move from one device to another, the same functionalities can be brought into the new device.

Components in our system are called facets[1]. Facets are hosted on facet servers. Clients request for facets from proxy servers. Proxy servers will return a suitable facet to the client, taking into account the resource requirements of the device and user preferences. There are execution servers around to provide a "computational grid" for devices to delegate execution of facets to. The client devices interact with each other in various ways. A client may get some data from a peer, or it may ask a peer to execute a facet for it. A client may also decide to move to another device, transferring its execution state and its data.

Facets

Facets are the units of composition with two essential features: (1) each facet carries out a single functionality and (2) a facet has no persistent state.

Functionality can be seen as a contract with clearly specified inputs, outputs, pre-conditions and post-conditions. Facets can be seen as components which implement these functionalities. They take in inputs and give the desired outputs, according to the terms of the contract.

Having a single functionality makes the components smaller, and also simplifies run-time composition. Since a facet has no persistent state, there are no dependencies between two calls to a facet. This makes the component throwable after use. Whatever is not needed can be thrown away, freeing up resources and memory for facets which

[1] The components were named "facets" because they are similar to facets of a diamond. Many small facets put together make up a dazzling diamond. In the same way, even though each facet may be small, when put together, they can create a very powerful application.

are currently running and to allow for other facets to be brought in. In addtion, clients are free to use other facets with the same functionality on the next call.

Facets may call upon the services of other facets to fulfill their contracts. Facet dependencies are the functionalities that a particular facet depends on. Two facets may have completely different implementations and yet achieve the same functionality. As long as they stick to the same contract, the facets are called compatible.

At runtime, a client will send a facet request which contains a facet specification to the network which includes the required functionality, runtime information and other requirements. The network will return a facet which matches that specification to the client. It is possible that each time, a different facet is returned for the same specification. Which facet is actually called can only be determined at run-time. The pictorial representation of facets actually executed at run-time is called the facet calling graph.

Active Facets are facets which are under execution at a given time, i.e. they are under use. Once a facet finishes its execution, it becomes inactive. Facets that are inactive can be discarded.

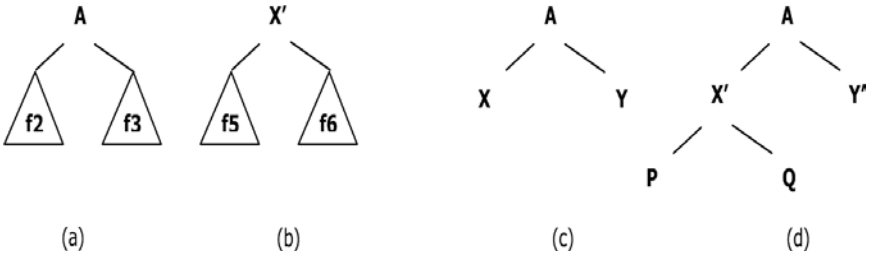

Figure 2: (a), (b) Dependency graphs of facet A and X'. (c), (d) Possible calling graphs for facet A

Facets are made up of two parts:

1. Shadow: A description of properties of the component including vendor, version, the functionality it fulfills, its dependencies and its resource requirements. It is used by the network infrastructure to locate the appropriate facet for the requesting device.
2. Code Segment: This is the body of the executable code, which implements the functionality. It follows the contract of the functionality.

Containers

Facets contain code segments and have a programming interface with which they communicate with each other. However, they cannot directly interact with the user. Containers act as bridge between the user and facets. A container contains routines for the user interface and a list of facet specifications. When a user carries out a certain action, it will give the corresponding specification to the client system to retrieve the first facet, which is usually called the "root facet".

Since facets have no persistent state, some state data is stored in the container, for example, the execution status of the facet and some shared data. They must be moved if execution is migrated to another device.

CLIENT SYSTEM

The client system plays a significant role in the whole architecture. Since it will be installed on the various clients, it has to be small enough to fit into devices with limited resources. It has to be able to support state capturing and migration mechanisms for supporting various types of mobility. Above all, it has to provide a dynamic execution environment to facets to come in, execute, and then be discarded.

Structure of the Client System

Since there are a variety of small devices with different OSes and execution supports, and the facets have to be executable on these various devices, the client system is developed on a virtual machine. This would guarantee portability of the facets among the heterogeneous devices.

A virtual machine is installed on top of the device operating system, over which the client system is implemented. The client system will accept facet specifications. It will then contact the network to request for the facets. Once it receives a facet, either from a server or a peer, it will load the facet and make it ready for use and return the ready-to-use facet to the caller. Once the facet is no longer in use, it is responsible for throwing away the facet.

The client system also handles all the background housekeeping, such as locating proxies and peers on the network, keeping track of the resources being used, and handling mobility.

Anatomy of the Client System

The core system consists of several modules, which interact with each other to provide the necessary functionality. They are briefly described as follows:

- Central Manager: The central entity which co-ordinates the activities of the various modules of the client system. It interfaces with the applications, containers and facets. It accepts requests for facets from the user-level, and delegates the appropriate tasks to the modules and returns a loaded instance of the facet. It also overlooks the main housekeeping of the client system.
- Discovery Manager: In an environment where the client devices are mobile, mechanisms are needed to locate nearby entities such as peers, proxies, etc. The discovery manager employs protocols to find devices in proximity to the client.
- Network Handler: The client device may employ different protocols and mechanisms to communicate with different entities, such as peers and proxies. The network handler deals with all the details of communication. It is responsible for making the connection and sending and receiving messages among the entities.
- Facet Loader: Facets are brought from the network into the client system. The facet loader loads and binds the incoming facets at runtime making them ready to use for the client program. It is also responsible for unloading facets that are not currently needed.
- Facet Cache: Some of the frequently used facets are cached instead being discarded. The facet then can be locally retrieved, instead of retrieving it from the network, improving the performance of the system. The facets in the cache can also be provided for use to peers. If a peer requests a facet and it is available in the cache, it can be sent to the peer.
- Lightweight Mobile Code Subsystem (LMCS): The LMCS is responsible for the mobility supports of the client system. More discussion of it is in the next section.
- Resource Manager: The Resource Manager ensures that there are sufficient resources for running the current application. It keeps track of the run-time resource usage of the client. It is also responsible for determining whether facets should be unloaded or removed from the cache. If it appears that there are not sufficient resources to run the next facet, it may contact the LMCS to delegate the execution of that facet.

INTELLIGENT PROXIES

In a pervasive computing environment, there could exist a large variety of components for different device configurations. It is not possible for developers to create components that are suitable for all devices due to the large variety of components for different device configurations. Therefore, the responsibility to find and return a component suitable for the device configuration should not rest on the users. The network itself has to have some intelligence in returning a suitable component for the client. However, returning a component suitable for the device configuration is still not enough in a pervasive computing environment which puts users in the center as the focus. The network should be able to tailor-make its response according to the particular user needs and preferences. The response should also be efficient irrespective of user movements to support the highly mobile nature of users. Thus, an intelligent proxy server is required, which will accept clients' requests, and respond to them efficiently according to the available run-time information, such as the device memory availability, as well as any user needs or preferences, no matter where they move to.

Traditional web proxies are not suitable for pervasive computing. They serve only as plain caching devices with the hope that what is in the cache will be used again in the near future and thus improving the access latency. They do not have the intelligence to locate a resource which will be most suitable for a client, and most of them do not consider the abstraction of a user. Recent proxies designed for pervasive computing (e.g. transcoding proxies (Bharadvaj *et al.*, 1999 ; Maheshwari *et al.*, 2002), QoS proxies (Nahrstedt *et al.*, 2001), have considered the client device's configuration. Some of them have also considered the preferences or needs for individual user. However, these proxies exploit the past request pattern of the individual user in order to pre-fetch the components for them. Efficient mobility support, which is important in pervasive computing, is also missing.

In our infrastructure, we propose a design for intelligent proxy servers that can act as matching and caching devices to find suitable facets for the clients. They have to achieve the following goals:

- Adaptability and customization: By allowing clients to specify queries instead of exact locations of resources, it is possible to choose a more suitable facet for the client. The proxies have to take into account details such as the client device configuration, current run-time state and user needs and preferences; and return a facet which is best suited for the particular device and user.

- Efficiency and mobility support: When a user is moving around, he may move from one proxy's area to another. The user should still be able to continue his work without any major disruption. Mobility support is therefore needed to give users a sense of "service mobility".

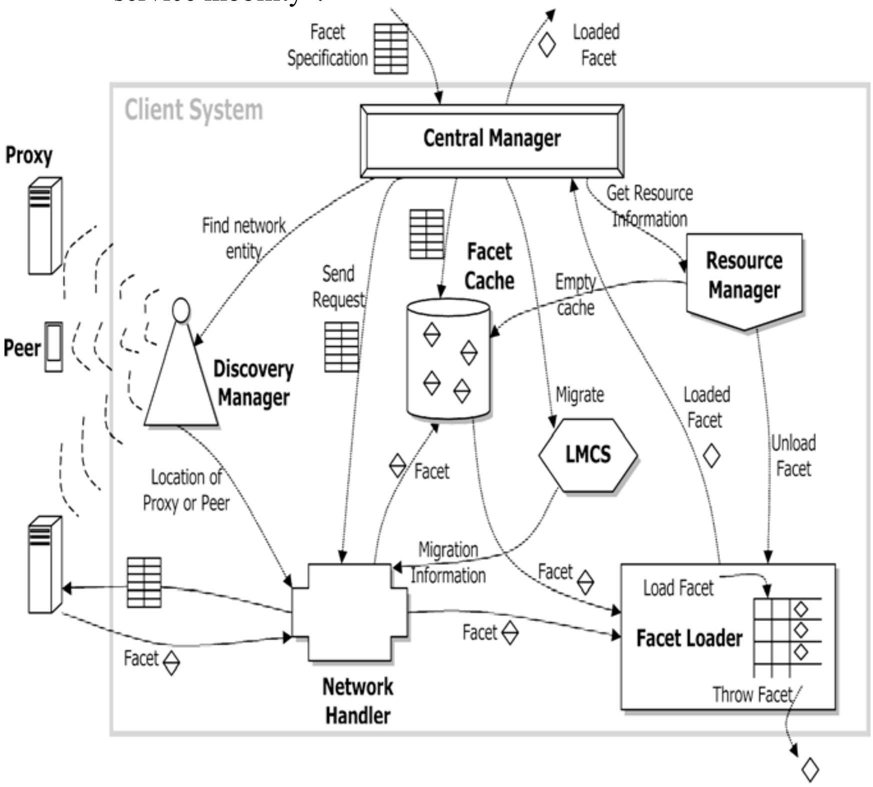

Figure 3: Overview of the client system

Features of intelligent proxy server

Below are some of the features of the intelligent proxy server:

- Dynamic service composition: Although the dependencies of the facets are pre-determined, the actual facets to be used are decided upon at run-time by intelligent proxies after receiving the request.
- Facet matching: Requirements specified in a request are a subset of certain facet shadows. In order to return a suitable facet for the client, subset matching is needed to match the requirements with the facet shadows. However, this kind of matching requires the

corresponding items to have the same value and is not powerful enough for our purpose. Range matching is also needed for some items, such as memory required, version number, etc.

- Facet pre-fetching for mobility support: In order to support mobility, intelligent proxy servers need to cooperate with each other so that user information, such as past request pattern, preferences, etc., can be moved to a new proxy as soon as the client moves into another area. In this way, facets needing to resume and continue the execution can be pre-fetched in advance.
- Each proxy server needs to maintain some prediction graphs, which are predictions of the calling graphs and built dynamically as facets are requested. By assuming that the order of the dependencies listed in the facet shadow follows the sequential calling order if the code is executed, facets to be pre-fetched are determined by a simple traversal of the corresponding prediction graph.
- For pre-fetching, a simple mechanism to look ahead on a certain number of facets is used. The proxy servers analyze the corresponding prediction graph for facets to be pre-fetched. It tries to pre-fetch the next facet that is most likely to be called by the facet being requested. The pre-fetched facet is then used for further analysis. The dependency graphs of the pre-fetched facets are used to supplement the prediction graph. This process continues until a certain number of new facets have been pre-fetched.

Design of intelligent proxy server

Each intelligent proxy server has four basic managers that work closely with each other: the matching manager finds a facet that best matches the client's specified requirements by utilizing the facet shadows; the pre-fetching manager pre-fetches facets into its local cache; the searching manager searches facets from other network entities; and the mobility manager is responsible for mobility support.

Figure 4 shows a typical client/proxy server interaction model, describing how a request is handled by an intelligent proxy server.

Besides normal interaction with the client, the intelligent proxy server also interacts with the facet servers periodically for any facet updates. This is done through the updating engine initiated by the searching manager.

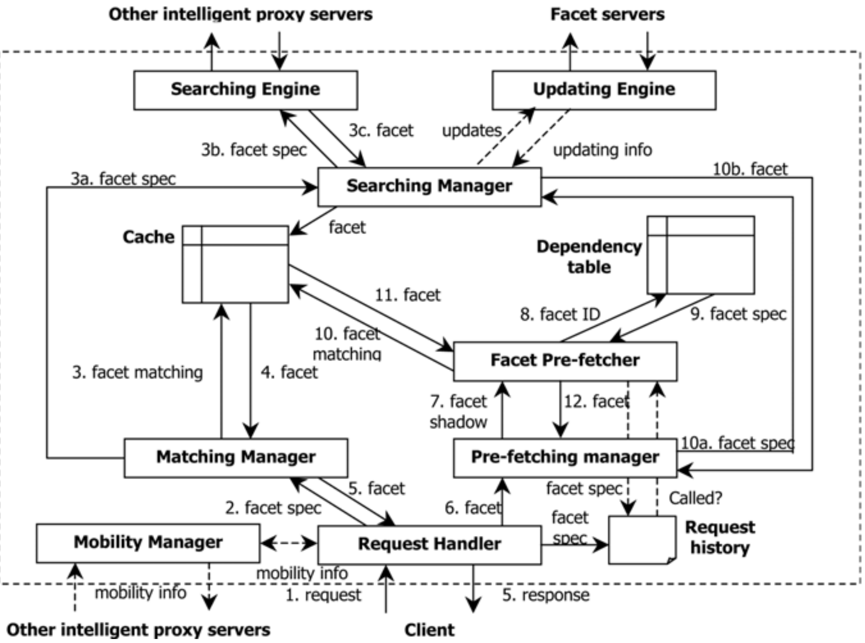

Figure 4: Typical interaction model of the proxy server

To implement mobility, client contacts the new proxy server when it enters another area. The intelligent proxy server then contacts some nearby intelligent proxies to identify the old proxy, and gets the corresponding user information, such as the past request pattern and the prediction graph, from it to continue pre-fetching the facets. After that, the new proxy analyzes the prediction graph to get the active facets, and sends a request to the request handler (which in turn delegates the responsibilities to other managers) for pre-fetching facets that might be used later.

Another possible mode of mobility is that user moves from one device to another. In that case, upon detecting the incoming client, the mobility manager would perform the same procedure as described above, but facets would be pre-fetched according to the new device's configuration.

LIGHTWEIGHT MOBILE CODE SYSTEM

As mentioned earlier, many applications in the pervasive computing environment will be user-focused. Users will access all sorts of information; applications will require a lot of user-data to tailor-make their services to meet the users' need. A lot of data will be moving through the infrastructure. However, there may be cases in which data cannot be moved, either because of security or privacy issues or perhaps it is just too large to move through a volatile low-bandwidth network. Since the data cannot move, we may need to employ some form of code mobility to carry out the needed computation.

Many times the devices would need to communicate with each other to provide their services. This is especially the case with intelligent appliances. They interact with each other directly and may form decentralized self-organizing networks among themselves. This scenario is not only limited to intelligent appliances. Other devices may want to share data without intervention from a central server. Thus peer-to-peer cooperative computing becomes an essential technology in a pervasive environment.

Another feature of the pervasive computing environment is user mobility – the ability for a user to continue a task as he changes his location or device. Traditionally, a client-server approach is used to achieve user mobility. A central server is used to store the information about the execution status of the task. When the user moves to another device, the execution status is recovered from the server. In an environment where there is limited connectivity, such an approach may not be feasible. It may be desirable for the corresponding code and execution status to be directly migrated to the destination node, without going through any central server.

There is no doubt that a flexible computation model is needed. Data and code mobility are believed to be the keys to enable it. We achieve that by incorporating the lightweight mobile code system (LMCS) into clients. With the LMCS, clients can adopt code mobility techniques to achieve their tasks.

The LMCS is adapted from mobile agent systems. We use lightweight mobile agents (LMAs) which are only responsible for migration. During migration, facet descriptions are plugged into an LMA and sent to the target. On reaching the target, the required facet is fetched and executed. This facet may call other facets, which are dynamically downloaded from the network, to carry out its function. The advantage of such a design over the traditional mobile agent is that an LMA can be very small. While traditional mobile agent has to carry the whole code with it when it migrates, an LMA only contains some minimal descriptions about its execution status. This effectively reduces the bandwidth requirement for agent migration.

In addition, such an LMA system is more flexible and dynamic. Other than the ability to travel on the LMCSs of different devices, an LMA itself has no functionality. It can be made to do different things by plugging different facets into it. For traditional mobile agents, this is not possible because once an agent has been coded, its "mission" remains the same, and their mission has to be determined at the time of agent creation. For LMAs, their mission can be plugged in as required at run-time, thus making the whole system more flexible.

As mentioned earlier, only the description of the facet is carried with the LMA and migrated to the target. The facet is then dynamically downloaded to the target. The proxies in the network ensure that the facet that is downloaded is the one that is best suited for the target. In other words, the migration automatically adapts to the target device. Such a scenario would be impossible with traditional mobile agents. They carry fixed code with them, and thus would only be suited for a uniform configuration of devices.

Most importantly, the LMCS and LMAs support peer-to-peer access in an elegant way with their inherent code-mobility features. Peers send request-response messages to each other. The difference between the conventional approach and the LMCS is that the former uses passive messages, whereas the latter uses an active entity (i.e., the LMA) for transmitting information between two end nodes. The conventional approach assumes the presence of some computing entity on the peer nodes analyzing and forwarding the incoming requests. On the other hand, like other mobile agent approaches, an LMA would actively find out what it needs on the peer node. If the peer node does not have what it needs, it would actively hop over to the next site according to its own itinerary. No external bodies other than the owner of the agent can dictate where the LMA goes to under normal conditions.

There are two important requirements in designing the LMCS. It should be small in size so as to satisfy the resource constraints of mobile devices. It should also support strong mobility, with which the execution status on the computational stack can be migrated to the remote site. With strong mobility support, states, data and codes now can move freely, thereby giving more flexibility to the computation model.

There are three core components in the design of mobility support: the container keeps the specification of the root facet as well as various execution states; the LMA migrates the container to the remote site following an itinerary; and the LMCS coordinates all migration activities.

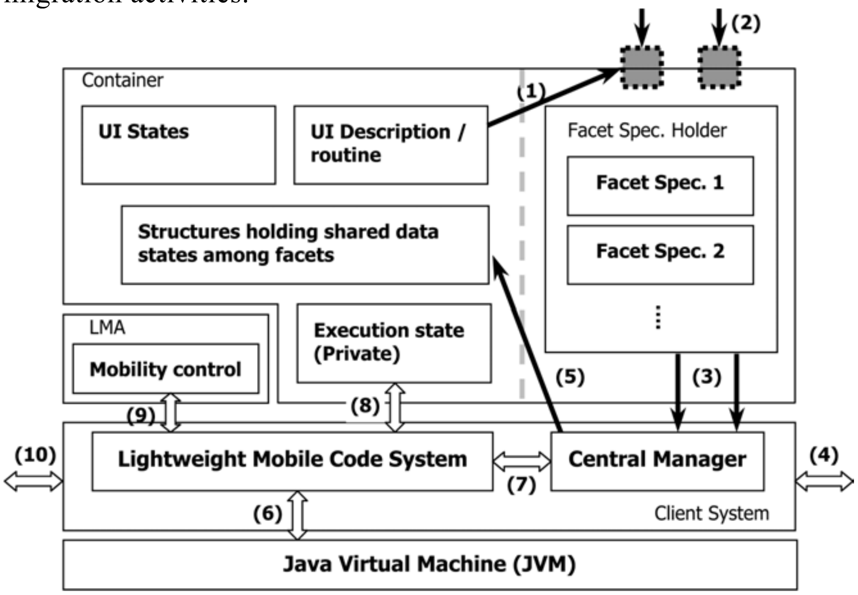

Figure 5: A typical interaction model in our mobility system

Figure 5 shows a typical scenario of facet execution and migration. When the container starts to run, the UI routine would be activated to bring up the corresponding user interface, and the user can interact with and execute a facet through that interface (1,2). When user issues a facet execution request through the interface, the central manager of the client system would fetch and load a copy of facet from its local cache or an nearby proxy according to the specification in the container (3,4). The facet would then be executed, during which some shared data states are saved into the container (5). Two possibilities of migration may now occur: either some migration routines inside the facets are triggered (proactive migration) (6), or the LMCS forces it to happen upon receiving migration request from the central manager (reactive migration) (7). In both cases, some migration information (e.g., the migration itinerary or destination) would be put into an LMA responsible for the migration (9). At the same time, the LMCS would ask the execution platform (e.g., a JVM) to pack up and put the relevant information of execution status in the container (8). Once the LMCS detects that all the information is ready, it would send the LMA, together with the container, to the destination site (10). On arrival at the destination site, similar but reverse operations would be carried out to recover the execution.

IMPLEMENTATION

Client System

We have built a simple prototype to illustrate the dynamic flow of facets. Our implementation is built on TransVirtual's KaffeVM on Compaq iPaq PDAs. We decided to use KaffeVM rather than J2ME (Java 2 MicroEdition) because KaffeVM has support for reflection and serialization.

The prototype was built on Compaq iPAQ H3800 with Intel StrongARM processor (206MHz), 64 MB SDRAM and 16MB Flash RAM. It was installed with CRL/OHH Bootloader v2.16.19, Familiar Linux v.0.5, Pocket Linux 1.0 with Kaffe Virtual Machine 1.06 (JIT) and pppd version 2.4.0b4.

The iPAQ was connected via the serial cradle to a Linux PC. A PPP connection was established over the serial line with the maximum bit rate of 115200bps. A web server has been set up on the Linux PC to act as the proxy server with very limited capabilities.

The Flash RAM is used to store permanent data. Familiar Linux and Pocket Linux took up 12MB which left us with 4MB of static memory for our system. The client system at present has a static size of 56KB. The JVM took 2.9MB of run-time memory, which is provided by SDRAM.

A facet is implemented as a JAR file. The shadow is an XML file in the JAR. The code segment is a package of class files with one of the classes being the main class of the facet.

Some preliminary testing has been carried out on the prototype. The latency to receive the facet after sending the request is somewhat constant, around 6.3 seconds. The time it takes to load a facet depends on the class size. For a 100KB facet, the average loading time is around 0.8 seconds. For a facet of 500KB, the loading time can go up to 1.5 seconds. We believe the performance can be much improved; work is underway to determine the bottlenecks.

To discard a facet, not only the objects have to be thrown away, but also the code of the facet loaded into the VM, i.e., the class code. The garbage collector usually collects objects which are of no further use; special techniques need to be employed for it to collect the class. We currently are considering two methods. One way is to use different user class loaders for each of the facets loaded in the system.

Another method is to use weak references. We are investigating the advantages and disadvantages of using both methods.

Intelligent Proxies

A simple prototype is being implemented as a proof-of-concept. Java is used for the implementation. The proxy server receives SOAP messages from clients, each containing an XML document outlining the facet requirements; and replies with a SOAP response containing the required facet.

The matching manager is being implemented based on XSet (http://www.cs.berkeley.edu/~ravenben/xset/), which is an XML database and query engine developed by the University of California that supports subset matching and range queries. We are modifying it to support facets instead of XML documents. For pre-fetching, a simple two-facet lookahead pre-fetching mechanism is being implemented. Its performance will be analyzed to see if more facets need to be pre-fetched.

Lightweight Mobile Code System

We use Java to implement the mobile code system. There are several advanages in making such a choice: it is platform-independent, which addresses the problem of running codes on heterogeneous devices; it has a built-in code-loading feature and provides codebase supports, which readily supports the mobility of code; and it also has the object serialization feature, which makes the transfer of data between two systems easy and convenient. However, using Java as an implementation language has a major drawback: the Java security policy forbids the dynamic inspection of the execution stack. In other words, strong mobilty of agents cannot be directly supported. Some special mechanisms to capture execution states are therefore needed.

Among several ways of capturing execution state in Java, we chose the source code instrumentation method. It adds in some migration instructions into the facets' source code before compilation. Compared to other approaches, this approach is much easier and is more well-suited to the pervasive computing world: it consumes less runtime resources when compared to the bytecode level instrumentation approach at class-loading time, and portability of code is also retained since the JVM is not modified.

One possible candidate for the source code instrumentation tool is JavaGo (http://web.yl.is.s.u-tokyo.ac.jp/amo), which is implemented by the AMO project group in Tokyo University. Its stated code size blow-up factor is about 20%, which is reasonable and affordable by some resource-limited devices. However, an additional pre-processor has to be added to it to suit our needs.

Applications

An image processing application was built in order to demonstrate the feasibility of building a normal application using the facet model (figure 6). The application consists of 15 facets, each providing a different functionality, out of which 10 were root facets (the grey ovals). The root facets essentially provide the functionality to open, to blur, to find edges and to flip images. The other facets provide functionalities such as matrix convolvers and converters. Further details can be found in a recent paper (Belaramani, et al., 2003). Another application we built was the Othello chess game (Kwan, *et al*., 2003). The game featured 95 facets supporting 19 different functionalities. There were 5 facets to each functionality, which provided the choices needed to demonstrate the functionality adaptation ability of the system.

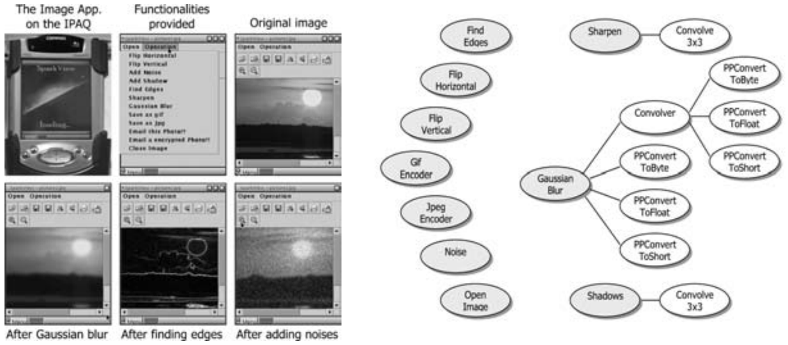

Figure 6: Screenshots and Structures of the Image Processing Application

RELATED WORK

Various projects for supporting component-based pervasive computing are discussed below.

Sun’s Jini (http://www.sun.com/jini/) provides protocols to allow services to join a network, discover what services are available in this network and dynamically access it through the use of Java RMI stub and its lookup service. However, Jini does not address the management of component-based applications and inter-component dependence. It only provides static look-up (exact matching) of services and does not consider the run-time resource constraints for small clients. Also, the large memory requirements imposed by Jini makes it not viable for most mobile devices. In addition, Jini announces service using UDP multicast by default, which may be suitable only in LAN-based application, but may not be applicable for large-scale deployment such as the Internet.

Berkeley Ninja (http://ninja.cs.berkeley.edu/) is a framework for dynamically composable wide-area services based on strongly typed reusable components. Ninja follows a dataflow-computing model. A group of dispersed services are identified and chained to form a path based on some resource demands. Client users are then able to obtain the required service by flowing data through the path. Ninja does not fully address code mobility. Mobile code is used only to instantiate the path. Also, all its reusable service components are not migratable. On the contrary, our design enables the dynamic loading of codes to client devices without moving client data for remote processing, unless the client is unable to handle large computations locally.

University of Washington's One.World Project (Grimm *et al.*, 2001) provides an integrated framework for building pervasive applications. One.World allows dynamic decomposition of applications into components and it separates the functionalities and data. We adopted the same approach on the separation of the functionalities and data. However, our facet is a Java-based component as we believe Java programming language is most platform-independent and is more portable and less complex in terms of engineering effort. Similar to the Ninja project, One.world did not address code mobility. A client-server model is adopted for obtaining Web services.

Illinois's 2K (Kon *et al.*, 2000) is a component-based operating system using CORBA as the communication mechanism. 2K supports dynamic resource management and automatic configuration in distributed environments. Also, 2K uses a prerequisite parser and resolver to fetch components and builds the run-time dependency graph. As the location of each service component is specified in the prerequisite specification, there is no need to do any dynamic matching, such as range queries or subset matching queries as in our intelligent proxy server.

Rochester Institute of Technology's Anhinga (http://www.cs.rit.edu/~anhinga/) is a distributed computing infrastructure designed specifically to support many-to-many distributed applications running on small mobile wireless devices. It addresses the lack of peer-to-peer support in wireless environment. It is built based on lightweight versions of Java and Jini. It uses a special M2MP protocol, which is a network protocol based on broadcast messages and uses Bluetooth for peer-to-peer communication. Such a broadcast approach may be difficult to scale up to fit the Internet and may even be wasteful in communication bandwidth. Instead, our design addresses the issue by sending out active LMAs following a pre-specified itinerary, thereby avoiding message broadcasting.

CONCLUSION

There is a trend towards a pervasive computing environment for everyone. Such an environment poses new requirements to software architechture design. The proposed infrastructure utilizes a combination of three methods to fulfill these requirements: firstly, a component-based development model; the components, known as facets in our system, are dynamically composed at run-time; secondly, intelligent network proxies that provide for efficiency, adaptability and mobility support; and lastly, lightweight mobile code systems to be installed in client devices to operate a flexible computation model. With this infrastructure, it is possible to perform computing and information access anytime, anywhere, from any device, and possibly for any application.

Acknowledgements: This project is supported in part by an Hong Kong RGC-CERG grant, "Content adaptation for mobile computing" and an HKU Research Seed Grant 28506002.

REFERENCES

A Lightweight Database for Internet Applications, UC Berkeley, http://www.cs.berkeley.edu/~ravenben/xset/

AMO Project Homepage: http://web.yl.is.s.u-tokyo.ac.jp/amo

Berkeley Ninja Project, http://ninja.cs.berkeley.edu/

Belaramani N., Wang C.-L. and Lau F. C. M. (2003), "Dynamic Component Composition for Functionality Adaptation in Pervasive Environments", Proceedings of 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS2003), San Juan, Puerto Rico, May, 226-232.

Bharadvaj H., Joshi A. and Auephanwiriyakul S. (1998), “An active transcoding proxy to support mobile web access”, Proc. 17th IEEE Symposium on Reliable Distributed Systems, Oct.

Cui Y., Xu D. and Nahrstedt K. (2001), “SMART: A Scalable Middleware solution for Ubiquitous Multimedia Service Delivery”, Proceedings of IEEE International Conference on Multimedia and Expo, August.

Fuggetta A., Picco G. P. and Vigna G. (1998), “Understanding Code Mobility”, IEEE Transactions on Software Engineering, Vol. 24.

Grimm R., Davis J., Hendrickson B., Lemar E., MacBeth A., Swanson S., Anderson T., Bershad B., Borriello G., Gribble S. and Wetherall D. (2001), “Systems directions for pervasive computing”, Proceedings of the 8th Workshop on Hot Topics in Operating Systems, May.

HKU Sparkle Project, http://www.csis.hku.hk/~clwang/projects/sparkle.html

Hussain S. and Mcleod R. D. (1998), “Personal Proxy Cache”, GRADCON’98 Proceedings, May 8.

Jini Network Technology, http://www.sun.com/jini/

Kon F., Campbell R., Mickunas M. D., Nahrstedt K. and Ballesteros F. J. (2000), “2K: A Distributed Operating System for Dynamic Heterogeneous Environments”, 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, August 1-4.

Kwan V. W. M., Lau F. C. M. and Wang C.-L. (2003), “Functionality Adaptation: A Context-Aware Service Code Adaptation for Pervasive Computing Environments”, Proceedings of 2003 IEEE/WIC International Conference on Web Intelligence, Halifax, Canada, Oct., 358-364.

Maheshwari A., Sharma A. and Ramamritham K. (2002), “TranSquid: Transcoding and Caching Proxy for Heterogenous E-Commerce Environments”, Proceedings of the 12th IEEE Workshop on Research Issues in Data Engineering, Feb.

Microsoft .NET Project, http://www.microsoft.com/net/

Nahrstedt K., Xu D., Wichadakul D. and Li B. (2001), “QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments”, IEEE Communications Magazine.

Neff A., “iAppliance 2001: You've Got the Whole World in Your Hand”, Bear Stearns & Co. Inc, March.

Oram A. (2001), “Peer-to-peer: Harnessing the Power of Disruptive Technologies”, O’Reilly & Associates, Inc., March 2001; ISBN 0-596-00100-X.

Sow D. M., Banavar G., Davis II J. S., Sussman J. and Rwebangira M. R. (2001), “Preparing the Edge of the Network for Pervasive Content Delivery”, Workshop on Middleware for Mobile Computing, Nov 16.

“Pervasive Computing” (1999), IBM Systems Journal, Volume 38, Number 4.

Rochester Institute of Technology’s Anhinga Project, http://www.cs.rit.edu/~anhinga/

Sun Open Net Environment (Sun ONE) Project, http://www.sun.com/software/sunone/

Szyperski C. (1998), “Component Software - Beyond Object-Oriented Programming”; Addison-Wesley / ACM Press; ISBN 0-201-17888-5.

The IBM's Mobile Aglets, http://www.trl.ibm.co.jp/aglets

Wang T. and Guan S. U. (2001), “Protecting Integrity for Code-on-Demand Mobile Agents in E-Commerce”, First International Workshop on Internet Computing and E-Commerce.