To appear in *Mobile Middleware*, Taylor and Francis Group.

Code-on-demand and code adaptation for mobile computing

Francis C.M. Lau, Nalini Belaramani, Vivien W.M. Kwan, Pauline P.L. Siu,
W.K. Wing, and C.L. Wang
Department of Computer Science, The University of Hong Kong, Hong Kong

ABSTRACT

Most commercial software packages present the user with a monolithic software program bundling many functions and features. The user pays for the unneeded features, and mobile devices have insufficient resources to cope. We introduce Sparkle, a proof-of-concept, mobile middleware for code adaptation using the code-on-demand design paradigm. To allow as many useful applications as possible to run on a mobile device, we propose changing the software development paradigm from monolithic chunks to small functionalities which can be dynamically downloaded on demand to the mobile device, and be disposed of afterward. An application could have unlimited functionalities which are selected on-the-fly based on the context.

1. INTRODUCTION

1.1. The Ultimate Thin Client

The ultimate mobile device should be thin, lean and mean. Being thin, it should be physically small enough to fit in a person's pocket. Being lean, it should only have those functionalities that do just what the user needs to do. Being mean, it is so affordable that one could replace the device without hesitation – to the extent that purchasing such mobile devices is as natural and convenient as buying a pack of beers. [1]

But is a thin-lean-mean device sufficient for all the computing needs of a mobile user in the future? The answer depends on what we do as users with mobile devices in the future. Advancements in technologies continue to give birth to more and more powerful mobile devices. Today we are witnessing the proliferation of very powerful handhelds blending everything – 3G phone, digital camera, PDA, media player, and massive storage – in a single device. And the trend seems to be that these devices would evolve to become more lightweight, cheaper in price, and stronger in communication.

In terms of computational power, however, we do not foresee the closing of the gap that separates the mobile devices from the PC. The latter has always to meet up with the rapidly growing software algorithm complexity, whereas mobile devices would tend more to target at applications at the lower end of the complexity spectrum. If mobile devices were to be used in place of the PC, even with the anticipated many folds increase in power within a short time, software performance would improve only along a plateau. The mobile device, after all, is bound by its form factor and limited power. Therefore, the wish that one day the mobile device can replace the PC as the ultimate personal computing device appears to be unrealistic. But what will a thin-lean-mean device be good for?

Mobile computing has created new usage paradigms. Tasks in the mobile computing world can be much more dynamic than those running on a desktop because of the changing context of the mobile user. Much research on location and context awareness is now taking place, which is irrelevant in the non-mobile computing world. The need for a mobile device to accommodate many dynamic tasks implies that the device must have been pre-installed with all kinds of software, which seems to go against the thin-lean-mean principle.

1.2. Ubiquity of Connectivity

Advances in communication are coming fast and strong. In a recent fourth-generation (4G) mobile communications field test, a maximum downstream data rate of 300 Mbps was recorded for a receiver in a car running at 30 km/h, and 800 m to 1 km away from the base station. Before 4G can be deployed, we are already enjoying the abundance of WiFi hotspots as well as the freest form of wireless communication via 3G. An emerging wireless technology is WiMAX which provides metropolitan area network connectivity at speeds of up to 75 Mbps and covering a practical distance of three to five miles. In a few years' time, the wireless network infrastructure will become completely mature offering an abundance of bandwidth and better coverage to connect any mobile device to the anywhere in the world-wide network.

Communication is probably the only parameter in the configuration of a mobile device that is not constrained when compared to a PC. In fact, there are all the reasons for making a mobile device more powerful in terms of communication capability than a desktop PC. If that will be the case, and when backed by a mature advanced wireless infrastructure, wireless devices will become very much an integral part of the global network, and many of the current network-based computing paradigms such as client-server and peer-to-peer computing will become applicable to wireless devices.

1.3. The Where, What and When of Computations

When user tasks are dynamic and diversified, it is infeasible to determine a priori which software to install in a mobile device. Ideally, new functionalities should be made available to the device when they are needed, or dynamically composed. There are four design paradigms related to this: client-server, mobile agent, code-on-demand [2], and remote evaluation. In the client-server paradigm, the client asks a server who implements the service to access some resources accessible by the server. We say that when someone implements the service, it holds the know-how. Remote evaluation is similar to the client-server paradigm but this time it is the client who holds the know-how which will be sent to the server to carry out the service. In the mobile agent paradigm, computing may be carried out by any device in a network. In the code-on-demand paradigm, a client device gets the know-how from a peer or a server and carries out the computation by itself.

In terms of capability, a mobile device may function between two extremes. In one extreme it acts as a remote display terminal; in the other extreme, it functions as a fully capable computational device. Remote display software, such as VNC [3], terminal services client, and g-cluster [4], etc. require a stable network connection between the client and the server. A stable connection, however, is not always achievable in a mobile environment. When wired or wireless network connections improve, it is likely that improvement is on the bandwidth but not the latency. For a user not to perceive any delay, real-time applications should not have a responsive latency longer than 50 ms [5], which is not easily achievable in a device-server operation mode where the connection could be multi-hop. When user commands and user screens are sent back and forth between the device and the server, an application would not be responsive enough.

The other extreme is to perform all functions in the device. In this role, the device is pre-installed with all the applications and data that are needed to handle all the user tasks. The problem is that it might not be feasible to determine the kinds of software to install a priori. Another problem is that the device might not have enough room to accommodate the entire collection of software.

An alternative exists in between the two extremes, which is that the mobile device would collaborate with some server or peers to carry out a user task. This would be more suitable in the mobile environment where user tasks are dynamic and device functionalities are composed dynamically.

A mobile device should be able to continue working when it is disconnected from the network. Therefore, client-server, mobile agent, and remote evaluation are not suitable for carrying out user tasks in the mobile environment. Code-on-demand is better because the program code is downloaded and executed in the mobile device. This paradigm would be more tolerant to various issues such as performance bottleneck, fault-tolerance, availability, service customization, user interface responsiveness, and device mobility. Therefore, in the context of mobile computing, code-on-demand paradigms are suitable for carrying out dynamic and diversified user tasks.

1.4. The Future

Much of the potential of mobile devices has yet to be exploited. Current mobile applications tend to be simple. Comparing the desktop version and PDA version of a word processor, for instance, we can see that a lot of the functionalities are not available on the PDA. Besides the small form factor which makes maneuvering difficult and hence some of functionalities not appropriate, there is no reason why feature-rich software has to be out of reach for PDAs in the future. We believe that even the most complex actions could be carried out on mobile devices by improving input/output modality and redesigning the software engineering approach.

So what can the thin-lean-mean device do for us in the future? In the not so distant future, it will probably play the role of a full-function computing but lesser device than the PC. Software or software components can be installed on the fly from a nearby server wirelessly upon request, which can be discarded after use. For this to be feasible, software must themselves be suitably "lean" so that downloading and installing them on demand will be efficient and that they will not take up more resources than what is commensurate with the user's needs. And lean software will naturally be more affordable – “mean”. In the longer term, as the wireless infrastructures around us become sufficiently powerful and stable, it can be envisioned that mobile devices can offload more or even all of their computations to the servers, and the devices will degenerate into very thin wireless remote display terminals. Users then will be free of all the trouble of managing a computing system at home or in the pocket. This chapter addresses the near-term solution of code-on-demand.

2. SMALL CODES FOR SMALL DEVICES

2.1. A New Notion of “Application”

Traditionally, applications are built as huge monolithic chunks. These applications provide lots of functionalities, yet too big to fit into a small device which is now prevalent in the mobile environment. The current solution is to develop other versions of an application that would fit, and such versions would most likely be downgraded, meaning that some of the functionalities are not available in these versions. If we compare a Word processor in a PC with the PDA counterpart, we would see that the functionalities of the PC version are overkill while the PDA version appears to be much deprived. Our solution to the problem is to build an application not as a monolithic chunk, but out of components that implement various functionalities needed by the application. The PC version and the PDA version could in fact draw from the same collection of components, and as such the idea of “version” becomes blurred.

A prototype to demonstrate our idea, called Sparkle, was introduced to support dynamic component composition and dynamic application reconfiguration. Sparkle is a component-based middleware for mobile computing. It enables application to be dynamically composed at run-time and reconfigured according to changes in the context. In Sparkle, applications are built from small functional units, called facets. These functional components could be implemented in different ways that fulfill the same functionality, and one of them is chosen at runtime based on certain contextual information. When an application runs, suitable functional units are downloaded from the network from a peer or from a server to the device. After the application has finished using the components, they may be cached or discarded. When a facet requires another facet, the latter would also be selected dynamically based on the context. Whenever the context changes because of changes in the environment, new functional components may be brought in to adapt to the environment. When an application is moved from one device to another, components of same functionality but using different implementations may be installed in the new device. Although the implementations are different, the same application will run and resume from the previous execution state. This is achieved by a mechanism that supports state migration of facets [6].

The dynamic composition and configuration of an application according to the context is a kind of code adaptation. An application consists also of a user interface (UI) and data or contents. By imposing a clear separation between them, adaptation can be applied to the code, the UI, and the data individually. The focus of the Sparkle project is on the adaptation of code. UI adaptation is temporarily handled via what is called a container. Contents can be considered a kind of data that is viewable, and we have done some preliminary work in contents adaptation, of which some of the techniques should be applicable to UI adaptation [7].

2.2. “Facets” of Functionalities

In Sparkle, applications are built from small functional units, called facets. The main purpose of the facets is to support dynamic component composition. Separation of functionality from data and user interface (UI) is the fundamental philosophy of the facet model. Applications allow users to carry out certain tasks. They provide certain functionality to carry out these tasks. These functionalities are embodied into facets.

Functionality is a single well-defined task in an application. The task could be as small as a matrix multiplication, or as big as detecting meaningful features of an image. It is mainly up to the programmer to decide what an application’s constituent functionalities are or how “big” they are. Given a set of inputs, the functionality determines what changes are made and the outputs attained. Essentially, functionality can be seen as a contract defining what should be done. The contract includes

1. the set of input parameters - i.e., the number and types of the parameters;
2. the set of output parameters - i.e., the number and types of the outputs;
3. description of what is carried out - i.e., what are valid outputs for a set of inputs;
4. pre-conditions, if any - for example, the ranges of input parameters supported;
5. post-conditions, if any - for examples, which values are nullified, and error conditions;
6. side-effects, if any - for examples, I/O, or changes to state in the “container”.

The contract defines the functionality to be achieved, but not how it should be achieved. Implementations can use different algorithms, each with different performance characteristics or resource requirements. As long as they abide with the contract, they can be considered to be achieving the same functionality. As a consequence, functionality defines the interface for interaction and is independent of the implementation. To make things simpler, every functionality is assigned a globally unique identifier, the functionality id (funcID). Thus, a functionality id (funcID) uniquely identifies a contract.

Facets are entities which implement the functionalities. They contain code components which follow the contract of their corresponding functionality. In our prototype, for simplicity, a facet implements only a single functionality. In other words, a facet cannot provide two or more functionalities. In future extensions, a facet may implement multiple functionalities that are related. This can achieve better scale of economy, and to the user, related functionalities should be loaded together anyway.

Given this limitation and the nature of functionalities, a facet has only one single programming entry point and is stateless. A facet being stateless means that a facet is independent of any previous invocations. Once the execution of a facet is finished, it is either discarded or is reset, so that they do not affect the execution of the next invocation of the facet. These features make facets throwable – a facet can be discarded from the run-time as soon as it is no longer needed. To keep some application states, an internal data structure called the container is used.

In short, the container is used to bring up an interface for user interaction, which in turn will request for the appropriate functionalities based on the user's input. A container provides a place for facets to run in. Each container is associated with a particular application and contains a set of functionalities which the application can offer. These functionalities are stored in the container as facet specifications. When a particular functionality is required, the corresponding facet specification is sent as a request to the proxy located in a server or a peer. It has a storage area to store the execution state and application data. By having such a storage area, facets may communicate with each other to obtain the application data they need. It also enables process migration when a user switches to another device.

Our approach differs from conventional distributed systems in which objects are used as a unifying abstraction for both data and functionality. Because functionality is bound with the specific data implementation it can act on, the object paradigm may not be a good fit for a mobile or pervasive environment [8-10]. Our design separates the data from the functionality so that it is possible to use different implementations of the functionality in different devices to operate on the data. It has also been argued that application functionality changes more frequently than data implementation and data layout. Therefore, it is preferable to store and communicate passive data rather than active objects. A clean separation between data and functionality allows them to be managed and to evolve independently. Therefore, we have separated data from implementation. We use the container to store the data.

Facets provide pure functionality. They take in some input; carry out their functionality resulting in the corresponding outputs. The user interface is just a means to access functionality. It is highly dependent on external factors such as display capabilities of the device and user preferences, rather than on the application or task at hand. Different UIs can be used to access the same functionality or task. In fact, the UI changes more often than the essential functionality of an application. Since the UI changes from device to device and version to version, it is desirable to keep it separate from functionality. As they are not bound to each other, this enables developers to change the UI without changing the functionality and vice versa, attaining a more intuitive and flexible software model.

2.3. Infinitely and Runtime Composable Software

A special branch of code adaptation is called functionality adaptation. Functionality adaptation involves changing the way the task is carried out in order to respond to changes. It selects different code implementations for execution depending on the context. For example, if a device does not have sufficient computation power, an application can execute another implementation of encryption using a smaller key.

Dynamic component composition provides a flexible mechanism for achieving functionality adaptation. Functionality adaptation is made possible by the following. First, the component model allows functionalities to be composed at run-time and discarded after use. Second, the context manager in a client device maintains information about the physical resource, network connectivity and the context of devices. Third, proxies match requests with suitable facets for clients to execute. A proxy is executed in a peer or a server to help the mobile device to choose a suitable facet. They are the main active entities for adaptation. Functionality Adaptation, in this approach, is achieved by choosing the appropriate component among different ones which have the same functionality.

Developers only need to specify which functionalities they require in an application, and provide different versions of them. The adaptation mechanism is transparent to the programmer. Which component gets picked depends on the system and the matching mechanism of the proxy. In addition, since applications are linked by functionalities, rather than specific components, as new technologies or devices emerge, developers only need to write a newer version of the affected functionalities. The proxies will automatically match these components under the appropriate conditions. Rewriting or reinstallation of the whole program is not required. Besides, since the components are thrown away at run-time after use, even the biggest programs can be used in a small device, depending on the size and the run-time behavior of each component.

During the course of application execution, a facet may call upon other facets to help achieving its functionality. To allow dynamic and flexible adaptation, facet providers do not specify the location about a requesting facet. Instead, they specify the functionalities that the facet requires for dynamic binding. These functionalities required by the facet are called its facet dependencies (or simply, dependencies). Facet dependencies, therefore, represent a local point of view of the facet. The dependencies that a facet depends on can be represented as a facet dependency tree, as shown in Figure 1(a). Facet dependency trees are only one level, as a facet only knows the dependencies it requires.

At run-time, when a facet requires another functionality for execution, the client sends a request for an actual facet of the required functionality. The returned facet, in turn, can have its own dependencies to help achieve its functionality. These dependencies are used as requests for the actual facets only when they are needed for execution. If we draw lines between a facet and the actual facets it calls at run-time, we come up with a facet execution tree. This facet execution tree cannot be determined statically, but can only be known at run-time. This is because different facets may be selected under different contexts. The facet tree shows the relationship between a facet and all the facets required at run-time for achieving its specified functionality, thus representing a global view of the facet. Figure 1(b) shows an example of a facet execution tree.

3. FUNCTIONALITY ADAPTATION

3.1. Context Sensitivity

In Sparkle, applications are able to take advantage of the context of the user, including location, the device being used, time, preferences and nearby services, to provide customized and relevant services to the user. For example, a facet for printing will be sent to a mobile device when there is a nearby printer. Such functionality is not present in the mobile device before the device approaches the printer. Furthermore, applications may adapt to the capabilities of the printer. For the same printing functionality, different facets may be implemented, one for monochrome output and one for colour. This way, different facets may be retrieved to the mobile device depending on the colour of the source image.

Our system is adaptive to four types of contextual change. The first type is device resources. These include factors internal to the device, such as the working memory available, the processing power, the energy, etc. The second one is network properties. These are changes in the network characteristics, such as the network bandwidth, network type, protocol, etc. The third one is environment. This includes factors in the surrounding environment, such as location, entities available nearby, time, etc. Finally, it is user preferences. These are specific choices which the user has made in relation to the execution of a particular application.

3.2. Portfolio of Functionalities

Software is commonly distributed to users, at least conceptually, as monolithic applications with a fixed set of functionalities. The major drawback of this model is the rigid boundary placed on the accessibility of functionalities. A functionality can only be accessed in the context of the application it is associated with, and not under the realms of another application.

We therefore introduce the concept of personalized software. Instead of having applications as the focal point of software development and distribution, software is treated as functionalities associated with and used by a particular user. Every user has its own portfolio of functionalities. The user decides which of the functionalities are needed and put them in the portfolio. Once in the portfolio, these functionalities are always accessible, in a sense blurring the application boundary.

The actual implementation of these functionalities is distributed at run-time. The functionalities are composed from various components, which are brought in at run-time and discarded after they have been used. The system plays the role of a corkboard, pinning up components when they are being used, and unpinning them when they are no longer required.

The advantage of such a model is its flexibility. Functionalities are not confined to being invoked only under the realms of certain applications. It facilitates the incorporation of new functionalities, updates to current functionalities and adaptation to new environments. The revenue model is a lot more accommodating to the specific needs of different users. In addition, our component based model allows for more code-reusability, and easier maintenance of the code-base for developers.

Sparkle is a demonstration of the feasibility of such a software distribution scheme. The basic foundation of Sparkle is facets which are software components used to build the functionalities.

Instead of having applications as the focal point of software and distribution, software can bee seen as functionalities associated with and used by a particular user. A user has a personal portfolio of functionalities. When functionality is needed, it is added to the portfolio of a user. When the user has finished using it, it is removed from the portfolio.

These functionalities are categorized according to the “functions” or “service” they provide to the user. Users can pick which functionalities they need from different categories to put into their portfolios. For example, a user is viewing an image, and decides to make changes to it. If a mobile device has not specified any image editing tool, the user should be able to go through a categorization process to add the tool to the portfolio and use it immediately. The image can then be edited regardless of the application context the image is open in. Such a scenario blurs the application boundary to a certain extent. Functions and tools are not bounded to a certain application, and therefore this enhances the productivity of the users. In addition, it is possible for the underlying system to have some intelligence or rules to predict what functions may be required in the near future based on the information of the functions the user called recently. These functions can either be displayed to the user, or brought in before hand to enhance the user-experience.

Even though the portfolio may contain several functionalities, these functionalities should be brought in as they are required. The underlying system should be like a cork-board in which functions are pined up and unpinned at run-time as needed. The functions should be brought in from the network, loaded, linked to the system, executed, unloaded and discarded at runtime. At start up, different users will receive different sets of components according to their portfolio. While carrying out computing, functionalities are brought in as they are required. There may be different versions of the same functionality suitable for different resource environments. The version which is brought in is the one most suitable for the current execution environment of the user. Once the functionality is used, it is discarded from the system.

Discarding a function from the system is not equivalent to removing a function from the portfolio. The portfolio of the user contains the list of functionalities that the user can access, rather than the actual components they are implemented by. When a function is discarded from the system, it can be brought in again from the network, when the user invokes that function again. All this is transparent to the user.

Basically, this takes modular programming a step further. Software is not only made of separate components, it is distributed separately as well. Programmers create components, each of which are small and carry out one thing well. They can create multiple versions which carry out the same functionality to suit different resource scenarios. Composers can leverage the different components to fulfill certain functionality, or to provide a group of functionalities. Users receive the components only when they are needed.

There is an irresistible trend towards mobile and pervasive computing. Users employ different devices such as PDAs and mobile phones to carry out computing. These devices are heterogeneous, limited in resources, and are connected to different network connects such as a wireless LAN or a Bluetooth ad-hoc network.

Because of the modular design and the fact that components are brought in only when they are needed, this demands a smaller working memory than big monolithic applications. Besides, only components which are suitable for the current computing environment are brought in, in essence achieving run-time compositional adaptation. Being able to throw away functionalities is important for resource usage efficiency. What is unwanted can be thrown away, freeing up resources for currently executing functionalities, or to bring in other functionalities. In effect, it allows small devices to run a group of functionalities which it would normally not be able to run had they been distributed in an application in the monolithic fashion.

Such a software architecture provides a convenient basis to enable context dependent applications. When a user moves from one place to another, say into a shopping mall, he may need to incorporate functionalities which are required to operate in that particular environment. For example, new functions could be incorporated which allow the user to securely book tickets in the cinema as soon as the user enters the mall, or perhaps to remotely order food in a restaurant and picking it up later. This may require the device to use proprietary protocols to talk to the shopping mall server. This protocol can be incorporated temporarily as functionalities provided by the shopping mall server, and discarded and removed from the portfolio when the user leaves the shopping mall.

3.3. Facet Architecture based on Ontology

Facet functionalities are specified by ontology. Ontology has been introduced for bridging the knowledge gaps between different domains [11]. Ontology represents the semantics of different concepts. It provides a formal, explicit specification of a shared conceptualization of a domain that can be communicated between people and heterogeneous application systems [12]. Ontologies for their applications are defined in the stationary environment. Thus, devices could only communicate using the same ontologies. Sparkle separates data and functionality. To describe the functionality, we use ontology to prescribe the semantics of the user-perceivable task description and provide a formal, explicit specification of shared conceptualization.

A facet consists of two entities: the code segment and the shadow. Code segment is the part where the executable code lies, and contains only one publicly callable method to be called by others. This code, when executed, should perform a predefined specific functionality.

A facet is described by meta-data called a shadow. This is used to identify a facet. A shadow contains the properties of a facet, such as the vendor, version, the functionality it performs, the resources it needs in providing the functionality, and the functionalities that the facet requires for execution. It includes information about the facet such as the function a facet provides, input and output specification, vendor and versioning information, resource requirements (e.g., device memory), functionally capability (e.g., rendering monochrome images) and its functionality dependencies (any other facets this facet would invoke), and the charging scheme for the use of facets. It is represented by an Ontology-based Task Description Language extended from the Web Ontology Language [13] and thus is human and machine readable. Figure 2 shows an example of a shadow. In the example, FlipVertical is a functionality of a facet and all facet vendors implementing this functionality uses this name in their shadows.

After facet functionality is described and the facet is located in a proxy, the proxy can use a two-phase adaptation technique to choose which facet should be sent to the client after the client has specified some functionality requirements. Clients do not need to rely on the servers for executing the services. The aim of the two-phase adaptation is to adaptively select a best suited facet from all available facets in a proxy. The first phase, called the filtering phase, is to filter the facets that satisfy the requirements of the client. These requirements include, at least, the functionality needed by the client and the amount of resources available in the client device for executing the specified functionality. All the facets filtered by the first phase have satisfied the client's requirements and are eligible for further processing. The second phase, called the selection phase, is to select a facet that best suits the device user. This decision is based on user preferences and other execution contexts of the client. The facet resulting from the two-phase adaptation is considered functionality-adapted and returned to the client. There are three key techniques in the two-phase adaptation: functionality filtering, resource filtering, and context selection. Functionality filtering ensures facets to achieve the functionality requirement of the client. Resource filtering ensures the functionality provided can be completed in the device, and context selection selects a facet that best suits the user and other execution contexts of the client. In order to allow more flexibility for the proxy system to choose among the facets, requests are specified in terms of queries instead of exact locations of facets. Furthermore, the proxy system maintains personal proxy caches for the users, so that facets can be better adapted to the device user. With all these supports, a good dose of adaptability can be provided by the proxy system.

4. DESIGN AND IMPLEMENTATION

4.1. The Sparkle Project

Sparkle aims to build an infrastructure that is suitable for pervasive computing environments. The infrastructure is based on the existing Internet infrastructure, with adaptability, mobility support and peer-to-peer cooperation as its main features. The adaptability feature addresses the problem of computing with heterogeneous devices in different execution contexts. Mobility support addresses the problem of continuing the current session in another device or at another location. Peer-to-peer cooperation avoids single point of server failure by allowing facets to be downloaded from nearby peers. In order to support pervasive computing, service implementation is in the form of a facet which is a mobile code component. Facets are downloaded on-demand to the client devices, executed and then discarded. In fact, when other facets are required in the course of program execution, they will be downloaded incrementally. The constituent code components of a service are not fixed at compile time, but are dynamically bound to form a service. Facets are mainly downloaded from proxies running in peers or servers. Facet servers, clients, and the intermediary proxy system are the three main components of Sparkle. Each of them plays a different role in the infrastructure. The following is a brief introduction; for further details, please refer to [14].

Facet servers are the places for storing facets. They are similar to existing web servers in that both are used as main storage servers that keep the up-to-date originals. They are used by the proxy servers for retrieving updated information. There is no restriction to the number of copies of a facet to be placed on these servers. A facet can be placed on more than one facet server, meaning that facets are not unique among the facet servers. Facets can be added to or removed from the facet servers by facet providers. These updates are usually quite frequent in terms of software maintenance.

In order to keep track of the updates made to the facets being stored, each facet server needs to keep a log of updates of the facets they store locally.

The clients are mainly mobile devices such as PDAs or mobile phones, although the computing model could be applied to non-mobile devices. Each of these computing devices has the capability of on-demand downloading, executing, and discarding the facets after use. This allows services of any sophistication to be executed on the client device. Whenever a service is required, a request is sent to a nearby proxy for a suitable facet. The request consists of a description of the required service, information about the resources in the client device that can be used for executing the required service, as well as some user information. A facet satisfying the request is then returned for execution. During execution, other sub-services might be required to help provide the service. Requests are then sent to the proxies when these sub-services are needed at run-time. This enables an unlimited chain of application functionalities.

The proxy system is a main component between the clients and the facet servers. It could be a mobile device peer or a dedicated server. Facets are cached in the proxies for fast retrieval and to reduce the workload of the facet servers. Client requests for facets are therefore sent to the proxy system instead of directly to the facet servers. Apart from being a caching device, the proxy system also acts as a recommender. It makes decisions on behalf of the clients and returns a suitable facet for each request according to the run-time execution contexts of the clients. Being able to choose a suitable facet for the client makes the proxy system intelligent. This enables the facets to be executed to provide the required services in the client devices, thus making the proxy system the key-enabler in Sparkle. In order to allow more flexibility for the proxy system to choose among the facets, requests are specified in terms of queries instead of exact locations of facets. Furthermore, the proxy system maintains personal proxy caches for the users, so that facets can be better adapted to the device user. With all these supports, adaptability can be better provided by the proxy system. On top of these, the proxy system also supports user mobility by cooperating with the lightweight mobile agent systems [15] in the client devices. With this support, the same user in a different location is treated equally, independent of the location and without affecting the computing experience. On-going services are, therefore, possible to be continued in another device with suitable facets being adapted. The proxy system also prepares the personal proxy cache to be used with suitable facets in supporting user mobility.

4.2. Universal Browser

The Universal Browser is not a traditional web browser. It is a browser designed for mobile environments. It invokes any function that a user wants on demand. It is a special graphical user interface (GUI) implemented in Sparkle.

This special graphical user interface allows the user to dynamically retrieve functionalities they want. As shown in figure 3, the user can use the Universal Browser to browse web pages, play games, and edit images. These functionalities are retrieved from the network when needed by the user. At startup, the device shows an "empty" GUI. Moreover, these functionalities could be thrown away after use to reclaim resources that may be needed by other functionalities. The Universal Browser, since supported by Sparkle, can help a user to find the suitable facet (i.e., functionality) that matches the device, and discard it when the facet is no longer needed.

Furthermore, the Universal Browser is a context-aware and extensible application. The context-awareness of Universal Browser is totally different from that of state management. The former is at the application level while the latter is at the system level. Context-awareness here means the downloading of different functionalities under different contexts. For example, facets that have higher memory demands are downloaded to the browser in a notebook PC, because they can render an image more quickly than the memory-thrifty ones.

Referring to the image viewer application, called SparkleView (which showing the puppy), in Figure 3, if the facet is cached on the device, the UI will indicate to the user that the functionality is available locally, and the icon's background changes to dark blue. If the facet is discarded by the underlying system, the UI will indicate it to the user accordingly, and the background color of the icon changes to light blue. SparkleView was run on a Pentium III Mobile CPU 1133 MHz with 384 MB RAM and Windows XP operating system, a configuration that is not much more powerful than some of the latest PDAs. The proxy had the following configuration: Pentium 4 2.26GHz PC with 512MB RAM and Fedora Core 2 operating system. The underlying Sparkle system is roughly 650 Kilobytes. The SparkleView "application" is 115 Kilobytes, most of which is the size of the user interface, and the facets are 44 Kilobytes in total. Figure 4 shows how long it takes to run different functionalities, which includes the time to bring in the required facets. It can be seen that Gaussian Blur takes a very long time. This can be attributed to the fact that Gaussian Blur has two levels of dependencies. It calls at least three other facets on every execution. Thus network delay to bring in the facets has an impact. Also, Gaussian blur involves a lot more mathematical calculations than the other functionalities and hence it takes more time.

5. RELATED WORK

A number of component-based middleware exist. Not many of them, however, consider resource constraints in mobile devices. It seems that the code-on-demand design paradigm to handle dynamic functionality composition is rare. In addition, most of them do not consider multiple implementations of the same functionalities [8]. Multiple implementations are necessary for different contexts, arising from such factors as device capability and user preferences. For example, a user wanting just a fast glimpse of an image would prefer a faster image rendering component that trades image quality. Even if multiple implementations are allowed, some of them require the implementations to be programmed as a single component [16]. This results in other parts of a component being superfluous and occupying memory resources. For some systems, adaptation is application specific [17,18] whereas we employ a system wide adaptation scheme in our middleware.

The Code Collection Project [19] shares our belief that software components for resource-constrained devices should be easily plugged and unplugged. They proposed an approach to optimize the use of memory via a garbage-collection-like algorithm.

The algorithm unloads (i.e., discards) methods that are likely to be not needed in the near future. Sparkle, on the other hand, focuses more on the "loading", for which there could be many choices for a requested functionality, and some components could be loaded automatically because of dependencies.

Open Services Gateway Initiative (OSGi) [20] is a service platform specification. Similar to our facet architecture, they use software components called bundles which are selected and downloaded to Internet appliances on demand at runtime. Bundles may also be removed after use. Although they have a bundle specification similar to our facet shadow, their specification is generally not migratable. This leaves the selection job to the server where a bundle is homed. In contrast, our model facilitates proxy based code-on-demand paradigm for better runtime performance. This enables peer-to-peer collaboration to share facets. Furthermore, our facet selection scheme is more flexible. In OSGi, bundles are selected based on a service ID and a predefined selection preference, whereas our facets are selected based on ontological specification and the range of functionalities a facet is capable to provide.

2K [17] supports reconfiguration of component systems at run-time. 2K is actually a distributed OS rather than a middleware. Similar to our model, when a component is needed in 2K, the component and its prerequisite components are brought in from the component repository, which may be located locally or in the network. After that, the resource manager is contacted to allocate the required resources for the components. Developers have to provide specialized components called configurator objects to handle dynamic reconfiguration. If any changes occur to the run-time resources, the resource manager will request the component configurator to make an adaptation. The adaptation policy, however, is application specific. Every application implements its own adaptation policy, which puts some burden on the application programmers. The facet model instead employs a system wide adaptation policy. The system has a better picture of the resource needs of all the running applications. This lets programmers focus on application logic rather than adaptation details, making it easier to develop mobile applications.

Aspect oriented programming (AOP) [21] is a programming technique to ease the development and maintenance of software. It facilitates common or similar program code pieces for cross-cutting concerns being neatly applied in appropriate places in different programs. For example, to optimize memory used when processing an image via a series of image filters, the filter codes typically are fused together into a compound code segment. In AOP, each filter is specified individually and then fused by a program called aspect weaver which generates the tangled code. Maintenance and management become easy because individual image filters are dealt with, and not the tangled code. Sparkle is similar in the sense that it fuses small components into tangled code. It is different from AOP however in that the fusion is not limited to components implementing cross-cutting concerns, and that the actual choices of components to be fused depend on contextual information. AOP complements other programming paradigms – for example, AspectCCM [22] uses AOP in their component model. Sparkle could benefit from AOP in the same manner.

Plug-in as has been popularized by Web browser software is a code-on-demand paradigm where new functionalities are downloaded on the fly and then executed. The fundamental difference between our model and plug-in is that in plug-in, the entire program binary for the plug-in is downloaded before execution happens, whereas our facets are downloaded one by one incrementally, and a facet can be extremely fine grained. Although Sparkle may cache or prefetch facets for better performance, it does not need to download the whole functionality once and for all. Besides, facets may be discarded after use to reclaim resources. Many plug-in implementations, unfortunately, require manual uninstallation. Furthermore, plug-in does not consider adaptation whereas Sparkle considers code adaptation based on contextual changes.

No-touch deployment [23] is another code-on-demand design paradigm where program assemblies are downloaded incrementally on demand. Similar to plug-ins, the exact locations of the assemblies are predetermined beforehand and therefore dynamic adaptation of functionalities based on contexts is not feasible.

6. ECONOMICS OF CODE-ON-DEMAND AND ADAPTATION

Under Sparkle, users only request for what they need. They receive in return an application with just the right amount of functionality, and extra functions may be added later. In fact, the concept of an application is blurred as there is no limit on the number of functions that could be added to an application. Because only the required functionalities are downloaded, users can pay for only what they actually use. This is fundamentally different from the current off-the-shelf software model where one pays for a large software package but ends up using only a small portion of its functionalities. Run-time installable components could come from any vendors so long as they are compliant with the corresponding facet contracts. To perform a user task, the user does not need to be concerned with which vendor provides the functionality. Rather, facets are retrieved based on the user context. Facet providers therefore would compete with each other to create high value-for-money facets that suit the users.

If software development adopts the Sparkle model, the software industry may benefit in a number of aspects. First, every vendor has an equal opportunity to reach their customers. This is because the customers would be looking for suitable facets and not necessarily a particular vendor. Facets are based on open standards – the contracts and specifications – thus creating an open field for any vendor capable of producing the right sorts of facets to set foot in. An actual running application would likely be composed of facets by a multiplicity of different vendors. Second, as applications are dynamically composed from facets, the same collection of facets could probably be used to produce, at a very late stage of the production cycle or even at run-time, different versions of an application suiting different computing platforms or devices. This is unlike the current practice where different versions of an application are predominantly standalone and self-contained pieces of software; any sharing of code would have been done in very early stages, and it is hard to add new functionalities to all the versions conveniently. This has a bearing on how vendors roll out new, killer functions to attract additional revenue. These functions have to be either bundled in a new version of the software, or adopt the plug-in model. Either way, it is not the best solution in terms of cost and user convenience.

Finally, with a Sparkle-like architecture, Internet services providers may take on a new business model which includes hosting a software repository for facets.

7. CONCLUSION

Sparkle is a code-on-demand adaptive mobile middleware that allows feature rich applications to run on resource constrained devices. This is done by introducing a new programming model where functionalities are made up of small functional units called facets. In the current implementation, each facet performs exactly one functionality and may be downloaded on demand and discarded after use. Because of this easy come-and-go mechanism for facets, an application may have an unlimited number of functionalities over time. This fundamentally changes the concept of conventional mobile applications that have bounded features. In addition, facets are not statically linked to an application. Facets are chosen based on various contexts such as environmental context and device context during the course of application execution. This means that different facets may be downloaded for execution in different contextual situations. This makes Sparkle adaptive.

Sparkle is an experiment for proving a concept, which inevitably has not touched upon many related important issues that would need to be addressed in a real design, including security and UI adaptation. Security could be a problem when we mix and match facets from diverse sources or vendors. A misbehaving facet could be most damaging if it has been adopted in the composition of a large number of applications. Some applications, such as computer games, have a complex UI – whether they can be composed from facets and how to synthesize the resulting dynamic UI are interesting questions for future research.

ACKNOWLEDGEMENTS

This research is supported in part by a CERG grant (A/C 10205435), “A holistic approach to structured web document viewing in small devices”, from the Hong Kong Government.

REFERENCES

1. M. Weiser, “*Computer Science challenges of the next 10 years*”. [http://www.ubiq.com/hypertext/weiser/UbiHome.html]
2. A. Fuggetta and G. Vigna, “Understanding code mobility”, *IEEE Transactions On Software Engineering*, Vol. 24, No. 5, May 1998, 342-361.
3. RealVNC. [http://www.realvnc.com/]
4. g-cluster. [http://www.g-cluster.com]
5. C. v. Hardenberg and F. Bérard, “Bare-hand human-computer interaction”, *Proceedings of the 2001 workshop on Perceptive user interfaces*, ACM Press, New York, 2001, 1-8.
6. P.P.L. Siu, C.L. Wang, and F.C.M. Lau, “Context-aware state management for ubiquitous applications”, *International Conference on Embedded and Ubiquitous Computing (EUC-04)*, Aizu, August 2004, 776-785.
7. W.Y. Lum and F.C.M. Lau, “User-centric Content Negotiation for Effective Adaptation Service in Mobile Computing”, *IEEE Transactions on Software*

Engineering, Vol. 29, No. 12, December 2003, 1100-1111.

8. R. Grimm, T. Anderson, B. Bershad, and D. Wetherall, “A system architecture for pervasive computing”. *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000, 177-182.
9. R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble and D. Wetherall, “Systems directions for pervasive computing”, *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001, 128-132.
10. R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall, Programming for Pervasive Computing Environments, Technical Report UW-CSE 01-06-01, University of Washington, Department of Computer Science and Engineering, June 2001.
11. J. Heflin, Web Ontology Language (OWL) Use Cases and Requirements, February 2004. [http://www.w3.org/TR/2004/REC-webont-req-20040210/]
12. T.R. Gruber, “A translation approach to portable ontology specifications”, *Knowledge Acquisition*, Academic Press Ltd., Vol. 5, No. 2, 1993, 199-220.
13. Web Ontology Language. [http://www.w3.org/2004/OWL]
14. N.M. Belaramani, Y. Chow, V.W.M. Kwan, C.L. Wang, and F.C.M. Lau, “A component-based software architecture for pervasive computing”, in *Intelligent Virtual World: Technologies and Applications in Distributed Virtual Environments*, World Scientific Publishing Co., 2004, 191-212.
15. Y. Chow, W.Z. Zhu, C.L. Wang, and F.C.M. Lau, “The state-on-demand execution for adaptive component-based mobile agent systems”, *Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS 2004)*, July 2004, 46-53.
16. S. Yau and F. Karim, “Component customization for object-oriented distributed real-time software development”, *In Proceedings of 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, March 2000, 156-163.
17. F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt and F.J. Ballesteros, “2K: A distributed operating system for dynamic heterogeneous environments”, *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00*), August 2000, 201-210.
18. R. Litiu and A. Prakash, “DACIA: A mobile component framework for building adaptive distributed applications”, *Operating Systems Review*, Vol. 35, No. 2, April 2001, 31-42.
19. L. Popa, I. Athanasiu, C. Raiciu, and R. Pandey and R. Teodorescu, “Using code collection to support large applications on mobile devices”, *MOBICOM 2004*, September-October 2004, 16-29.
20. Open Services Gateway Initiative. [http://www.osgi.org]
21. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, J. Irwin, “Aspect-oriented programming”, *Proceedings of European Conference for Object-Oriented Programming*, Springer-Verlag, 1997, 220–242.
22. P.J. Clemente, J. Hernández, J.M. Murillo, M.A. Pérez and F. Sánchez, “Component-based system design and composition: an aspect-oriented

approach", In K.K. Lau (Eds.), *Component-Based Software Development: Case Studies* (Chap. 5, 109-128), 2004.

23. No-Touch Deployment in the .NET Framework. [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbtchNo-TouchDeploymentInNETFramework.asp]

ABOUT THE AUTHORS

Francis C.M. Lau received his PhD in Computer Science from the University of Waterloo in 1986. He is currently an associate professor and head of the Department of Computer Science at The University of Hong Kong. His main interests are in parallel and distributed computing, Internet and the WWW, mobile computing, operating systems, and computer music. Contact him at fcmlau@cs.hku.hk.

Nalini Belaramani received her Master of Philosophy in Computer Science from The University of Hong Kong in 2002. During her study, she designed the Facet programming model for the Sparkle project. She is now a Ph.D. student at the University of Texas at Austin. Her current interests include distributed systems and autonomic computing. Contact her at nalini@cs.utexas.edu.

Vivien W.M. Kwan received her Master of Philosophy in Computer Science from The University of Hong Kong in 2002. She developed the intelligent proxy system for Facet-based software architecture of the Sparkle Project. Contact her at vjwmkwan@cs.hku.hk.

Pauline P.L. Siu received her Master of Philosophy in Computer Science from The University of Hong Kong in 2004. She developed the context-aware state management system (CASM) for the Sparkle project. Contact her at plpsiu@graduate.hku.hk.

Wai-Kwong Wing received his B.S. degree in Computer Science and Information Systems from The University of Hong Kong in 2001. He is a Ph.D. candidate in The University of Hong Kong. His research focuses on pervasive computing. Contact him at book@wkwing.com

Cho-Li Wang received his B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985. He obtained his M.S. and Ph.D. degrees in Computer Engineering from University of Southern California in 1990 and 1995 respectively. He is currently an associate professor of the Department of Computer Science at The University of Hong Kong. His areas of research include parallel architecture, cluster and grid computing, mobile and ubiquitous systems. Contact him at clwang@cs.hku.hk.

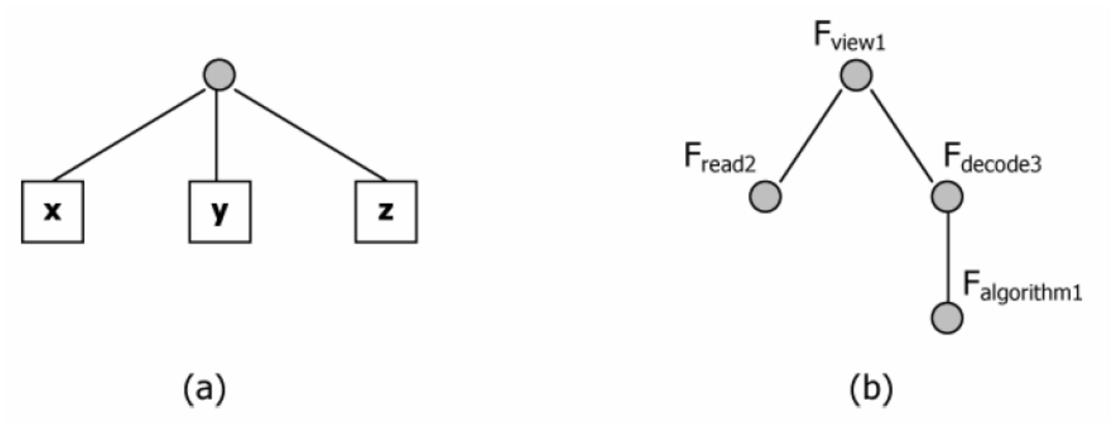

Figure 1: (a) A facet dependency tree; (b) a facet execution tree.

```
<?xml version="1.0"?>
<facet>
        <identifier>FlipVertical</identifier>
        <functionality_id>FlipVertical</functionality_id>
        <vendor>Sparkle</vendor>
        <version>
                <major>1.0</major>
                <minor>a</minor>
        </version>
        <resource>
                <memory>
                        <static unit="kbytes">2</static>
                        <dynamic unit="kbytes">40</dynamic>
                </memory>
        </resource>
        <dependencies></dependencies>
</facet>
```

Figure 2: A shadow example.

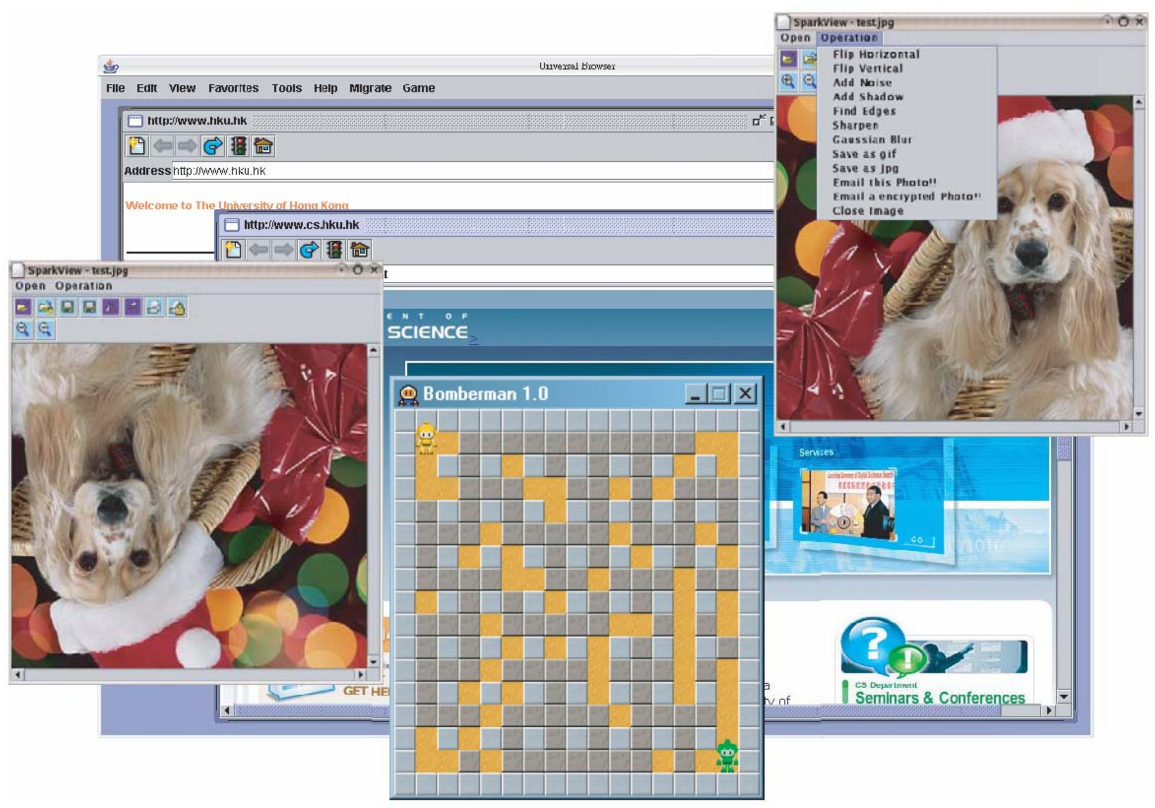

Figure 3: A screenshot of the Universal Browser.

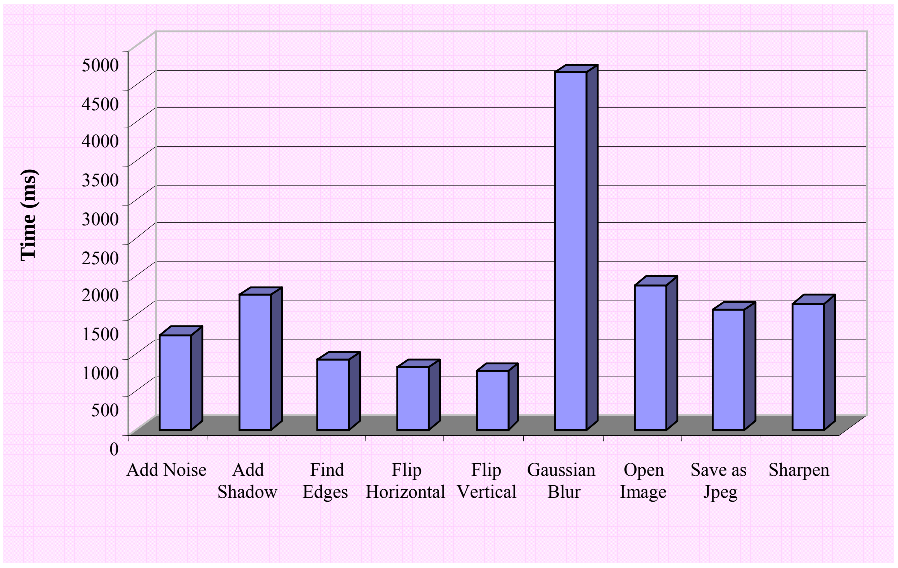

Figure 4: Invocation of different functionalities.