A COMPONENT-BASED SOFTWARE SYSTEM WITH FUNCTIONALITY ADAPTATION FOR MOBILE COMPUTING

BELARAMANI NALINI MOTI

M. PHIL. THESIS

THE UNIVERSITY OF HONG KONG
2002

A COMPONENT-BASED SOFTWARE SYSTEM WITH FUNCTIONALITY ADAPTATION FOR MOBILE COMPUTING

by

Belaramani Nalini Moti

B.Eng. *H.K.*

A thesis submitted in partial fulfillment of the requirements for

the Degree of Master of Philosophy

at The University of Hong Kong.

August 2002

Abstract of thesis entitled

A COMPONENT-BASED SOFTWARE SYSTEM WITH FUNCTIONALITY ADAPTATION FOR MOBILE COMPUTING

Submitted by

Belaramani Nalini Moti

for the degree of Master of Philosophy
at The University of Hong Kong
in August 2002

All things are affected by change. This is especially true for mobile computing environments. Everything, from devices used, resources available, network bandwidths to user contexts, can change drastically at run-time. It therefore becomes imperative for software systems and applications to be able to adapt to these changes in order to provide a suitable and relatively stable working environment for users.

Various techniques of adaptation have been researched, for instance changing the quality of data accessed, or changing routing information dynamically. These each addresses a certain aspect of change affecting the computing environment. However, dynamically changing how an application carries out its functionality – functionality adaptation – has not been sufficiently explored in the context of mobile computing. Techniques do exist, however, with limited flexibility and adaptive capability.

My work is motivated by the desire to devise a flexible and intuitive functionality adaptation technique, which can adapt to many different types of change affecting a mobile computing environment. The basis of this dissertation is dynamic component composition. Software and applications are made up of components which are assembled at run-time as they are required. There may be several components carrying out the same task. Which component is used for that particular task depends on the run-time execution environment. Under different run-time conditions, different components are used. Each of these components may have different run-time characteristics and adapt the execution of the task at hand, thereby achieving functionality adaptation. A new component model, the facet model, has been designed in order to realize the above technique.

The fundamental philosophy of the facet model is the separation of functionality from data and user interface (UI). Applications can be broken up into facets along the lines of

functionalities. Every facet provides a certain functionality. There may be more than one facet achieving the same functionality. At run-time, the appropriate facet is brought in and executed. This enables applications to be linked by functionalities, rather than by exact components. Functionality adaptation, in this approach, is achieved by choosing the appropriate component from several components with the same functionality. The adaptation mechanism is, in fact, transparent to the programmer.

This thesis classifies into categories the different types of adaptabilities that mobile systems exhibit, and the different types of techniques employed to achieve them. It defines functionality adaptation and argues that dynamic component composition is a flexible mechanism to achieve it. Finally, it looks into the details of the facet component model – a component model especially designed for dynamic component composition in the context of mobile computing – and illustrates its feasibility and applicability through the implementation of the Sparkle software system.

To Kiran, Mom & Dad
for their unfaltering support.

Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Signed . . . . . .

Belaramani Nalini Moti

Acknowledgements

I must thank, first and foremost, my supervisor Dr. Cho-Li Wang, without whose guidance and patience, this dissertation would not be possible. His enthusiasm was one of my main motivations for pursuing this dissertation. I am also grateful to my other supervisor, Dr. Francis Lau, who at various points of my research enabled me to step back and re-evaluate where I was heading.

I must thank my project mates, Chow Yuk and Vivien Kwan. Working with them, made research fun. Their advice in all the discussions we had has played an important role for the realization of this thesis.

Research would have been very daunting if it not were for my officemates, Cathy Luo, Fang Wei-Jian, Felix Cheung, Reynold Cheng, Zhuo Ling, and Dave Towey, who livened things up with their chatter, wit, and entertaining tales.

I must thank the technical staff and the administrative staff of the Department of Computer Science and Information Systems, who were extremely helpful and made sure everything went smoothly.

I must say, I have been blessed with lovely friends. I really appreciate Vivian Mak's and Henry Cho's heart-felt concern for my well being. Every meeting with them is always filled with laughter at top-volume. I must thank Ah Lam for always being there whenever I have had any problems, from installing Linux to deciding where to go for hiking.

I thank Manoj Mahboobani for actually helping me proof read my initial draft without having any idea about what was going on and Baggy Sartape for listening to my useless chatter, showing me the lighter side of things and helping me conquer bugs with a laugh.

I am deeply indebted to my closest friends and my pillars of support, Sunita Budhrani and Sabrina Kriplani. Life has a more entertaining and "lively" dimension with them around.

The best thing about them is that they go through the torture of standing my whims and fusses every day without saying a word, such as being forced to proof read my whole dissertation.

My sister, Kiran, is where my strength lies. She has been beside me through thick and thin, heard my endless chatter about computers at wee hours in the night, pretending to understand every single bit of it. She has made me laugh and made me mad. Talking to her just makes all my frustrations disappear. I don't think I would be the person I am without her.

And, of course, I thank my parents for being there for me, giving me all the comforts I need, and showering me with their love and care.

Finally, I thank the Lord Almighty, by whose grace I am where I am now.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"*Everything is in constant flux.*" – Heraclitus (c.540-480 B.C)

Nothing could be more true than the fact that all things are flowing – everything is under constant change. Even in the field of computing, we are witnessing great changes right before our eyes. When it first began, computing was limited to huge machines operated by scientists in research laboratories. At present, computing covers a lot more devices, such as PCs, laptops, and are used by a lot more people, even by those who don't have any technical background.

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 (PDA's) and mobile phones, taking advantage of wireless networks to connect to the Internet to provide some sort of services to the user.

The trend towards mobile computing should not be ignored. Like every other computing paradigm, mobile computing has its features and limitations. However, what sets it apart from the other paradigms is the amount of change it is affected by. Everything, from devices used, to resources available, to network bandwidths, to user context, can change drastically at run-time. From a computing environment's perspective, we could say that things in mobile computing *flow more quickly and in more directions* than traditional computing environments.

It, therefore, becomes imperative for software systems and applications to be able to adapt to changes, in order to provide a suitable and relatively stable working environment for users. Various adaptation techniques have been previously explored – from lower-level techniques of dynamically changing routing information, to changing fidelity (i.e. quality) of data. However, dynamically changing how an application carries out its functionality, functionality

adaptation, has not been sufficiently explored, in the context of mobile computing. Techniques do exist; however, with limited flexibility and adaptive capability.

My work is motivated by the desire to devise a flexible and intuitive functionality adaptation technique, which can adapt to a lot of different types of change affecting a mobile computing environment. The focus of my work is dynamic component composition. The basic philosophy is as follows: Software and applications are made up of components. The components are assembled at run-time as they are required. Which components are used to achieve a certain functionality depend on the current execution environment. Under different run-time conditions, different components will be used, hence achieving functionality adaptation. These components are brought, either from dedicated servers or from near-by peers. They are then linked to the run-time system and executed. Once the components are used, they are unlinked from the run-time system and thrown away, achieving memory efficiency.

A prototype architecture – the Sparkle architecture, was built to demonstrate the feasibility and applicability of the above proposed technique. A new component model – the facet model, was designed from scratch with innate support for dynamic component composition. The facet model is, in fact, of great significance since it is the means for applications to achieve functionality adaptation.

The fundamental philosophy of the facet model is the separation of functionality from data and user interface (UI). Applications can be seen as a means by which users perform tasks. An application usually provides several functionalities which users can invoke to fulfill their tasks. It is argued that functionality that an application provides changes more often than the data implementation and data layout. In addition, the UI can be considered just as a means to access functionality. Often, the UI changes more often than the essential functionality of an application.

Facets are pure functional units. Applications can be broken up into facets along the lines of functionalities. Every facet provides certain functionality. There may be more than one facet which fulfills the same functionality. As mentioned earlier, at run-time, the appropriate facet is brought in and executed. Hence, it can be seen that *applications are linked by functionalities, rather than by exact components*. Functionality adaptation, in this approach, is achieved by choosing the appropriate component among different ones which have the same functionality. This choice is made by the underlying Sparkle system and the various network entities. The adaptation mechanism is transparent to the programmer.

The main advantage of having this transparency is that it makes it a lot easier to build applications. Programming with support for adaptation for a huge variety of devices available can become a burden for programmers. Moreover, applications only possess a local view of the whole system. The resource manager has a global picture and thus is more suitable to do resource allocation and adaptation. Providing programmers with transparency for important system functions is not something new. Frameworks for Enterprise JavaBeans, CORBA and .Net give programmers transparency for persistence, transaction, etc. Transparency for adaptation can be seen as the next step in the same direction.

This thesis describes dynamic component composition and its use in achieving functionality adaptation in the context of mobile environments. It is organized as follows.

Chapter 2 looks into the mobile computing paradigm. It discusses its features and requirements. It also looks into the types of change affecting the environment, and the adaptation techniques which have been employed in order to respond to these changes. The main role of Chapter 2 is that it differentiates between adaptability and adaptation, and categorizes different types of adaptability and adaptation techniques. It discusses functionality adaptation in detail and points out the flaws of some of the current functionality adaptation techniques.

The focus of Chapter 3 is components. It looks into component-based technology in general, and looks into common current component models. It provides the background needed for understanding components, and dynamic component composition.

Chapter 4 describes the approach we have adopted in the Sparkle Mobile Computing Environment. It introduces the concept of dynamic component composition. It compares dynamic component composition with the traditional monolithic approach of distributing software and demonstrates how dynamic component composition is more suitable in a mobile computing environment and for functionality adaptation. This chapter also describes the overall architecture of the Sparkle system and the facet model – the component model we have devised for dynamic component composition. It describes the basic motivation and design philosophy of the facet model and finally compares the facet model with the other component technologies.

Chapter 5 describes the Sparkle client system which was built to support dynamic composition of facets. It looks into the constituent entities of the client system and how they have been realized in implementation.

Chapter 6 describes facets in greater detail. It provides a programmer's view of facet-based development. It describes how facets and their related abstractions should be implemented. Most importantly, it highlights the difference between facet-based development and object-oriented development.

Chapter 7 reports the experiments carried out on the Sparkle system in order to demonstrate the feasibility of the facet model. The experiments aimed to demonstrate the ability of the client system to support dynamic facet composition, to establish factors which affect the performance of the client and lastly, to show the feasibility of building a real-world application built by facets.

Chapter 8 takes a step back and looks at the overall picture. It addresses issues such as web services and the transparency of the facet model. It also looks into the deficiencies of the facet model and the Sparkle architecture.

Chapter 9 concludes the whole dissertation and points out avenues for future work.

In short, this dissertation makes the following contributions to research in field of Computer Science:

- It provides a classification of the different types of adaptabilities mobile systems and applications exhibit and also another of the adaptation techniques employed.

- It introduces functionality adaptation and its importance to mobile computing.

- It proposes dynamic component composition as a means of achieving functionality adaptation.

- Most importantly, it defines the facet component model, designed for dynamic component composition.

- It illustrates the facet model's feasibility and applicability in a mobile environment via the Sparkle architecture.

Chapter 2

Mobile Computing & Adaptation

There is an indisputable trend today towards mobile computing. Fuelled by the plethora of small, light-weight devices and the advances in network connectivity, especially in the wireless domain, one can easily predict that this is just the beginning of a whole new model of computing. This new model brings with itself, new features and places new requirements on software systems and infrastructures. What makes computing in the new model so difficult is its inherent and incessant change. The purpose of this chapter is two-fold. It aims to provide a brief introduction to the field of mobile computing and it explores the various dimensions of change which affect mobile environments.

This chapter is divided into two sections. The first section provides a brief overview of the mobile computing arena. It is not meant to be a comprehensive discussion of all the issues involved but it does provide an insight to the field. It first introduces mobile computing including its definition. It then discusses the typical features of the mobile computing environment and the requirements these features put on the infrastructure for support.

The second section starts off by exploring the kinds of change which affect mobile environments. It then differentiates between adaptation and adaptability, providing definitions, classifications and examples of both. Finally, it concludes that functionality adaptation in the context of mobile computing is not very well researched and that there exists a need for a system which has flexible and comprehensive support for it.

2.1 MOBILE COMPUTING

2.1.1 The Emerging Trend

Computing is no longer limited to a "computer" *per se*. Increasingly, many different types of devices are taking advantage of wireless networks and the Internet to provide services to the user. It has become commonplace to see cellular phones which allow web browsing and email reception with wireless connectivity, personal digital assistants (PDAs) with scaled-down versions of applications, digital video cameras with Bluetooth connectivity. Public places such as coffee shops and corridors of major commercial buildings have set up wireless networks for patrons to use.

All this indicates the advent of the new model of computing. With the increasing availability of small, light-weight and portable devices together with the advances in wireless technology, users are demanding mobile devices to provide them with more convenience and functionality. Users want to be able to access information and carry out tasks as they are moving from place to place regardless of the device they are using, may it be their PDAs, mobile phones, or even perhaps their watches.

Not only that, applications should be able to take advantage of facilities and information in the surrounding environment to provide relevant services. For example, a user may be working on a document on his PDA in a coffee shop. He should be able to use a printer available in the coffee shop directly, without having to go through the setup procedures. The mobile computing infrastructure should enable the discovery of nearby devices, services, etc., and provide links to them. In short, users want a seamless computing environment in which their device or their location is not a major concern when compared to the task they want to achieve.

2.1.2 What is Mobile Computing?

Mobile computing no doubt implies mobility. In this context, mobility can be seen in two dimensions: One dimension is *device mobility* - a user carrying out his tasks on a device while moving about, taking advantage of wireless connections. The other dimension is *user mobility* – allowing users to move from one device to another and still being able to carry out their tasks, access their information and continue where they left off. Device mobility, thus,

implies change of spatial co-ordinates, whereas user mobility implies change of computing device. At a software level, mobility can denote the movement of data and or the movement of code. To a certain extent, data mobility and code mobility are some techniques which can be employed to achieve user and device mobility.

What exactly, then, is mobile computing? Defining mobile computing is just as difficult as defining traditional computing. With the plethora of devices, technologies and applications employed, it is difficult to assign the definition of mobile computing to a single class of devices or applications. Chlamtac and Redi [14] have defined a mobile computing device simply as a *"computing device which can communicate through a wireless channel"*. Mobile computing can be seen as *the use of mobile devices and not-so-mobile devices, taking advantage of wireless means, to create a seamless computing environment enabling access of information, computation and co-operation.*

Many times the term "mobile computing" is used in conjunction with "pervasive computing" or "ubiquitous computing". Pervasive (or ubiquitous) computing aims to create an environment in which computing is so naturalized that people do not realize they are using computers[66]. This is achieved by deploying a lot of *smart devices* in working and living spaces that coordinate with one another in order to provide an intuitively personalized and constantly available system to users. These devices are small, may be mobile or non-mobile and embedded in gadgets such as appliances, cars, badges, etc., communicating through wired or wireless means. No doubt, pervasive computing and mobile computing have a lot in common. The difference lies in the focus of the two fields. Mobile computing has its focus on providing mobility especially by wireless means. Pervasive computing aims to fill the living environment with smart devices. Both fields complement each other to a certain extent. As mobile devices become smaller, and as the use of wireless technologies become more pervasive, the distinction between the two will become blurred.

2.1.3 New Environment, New Features

The mobile computing environment brings about new styles and scenarios. Users are moving in, about and out of the network, using different devices at different times, carrying out tasks or accessing information. The devices may be communicating with servers on the Internet or communicating with one another – forming peer groups, sharing resources, or services. It is evident that the mobile computing environment is very different from the traditional networked environment.

The essential features and demands, which set the mobile computing environment apart, include:

- *Device Heterogeneity*

  Instead of mainly PCs, computing is carried out on a wide range of devices, from laptops to PDAs, from mobile phones to pagers. Many of these devices are low-cost, small size, low weight devices. Each of them has very varied capabilities including display size, memory size, processing power, etc.

- *Device Resource Limitations*

  Miniaturization of devices is one of the propelling factors for the trend towards mobile computing. The compactness of the devices makes them limited in terms of memory, processing capabilities and, especially, power. No doubt, advances in technology will increase the amount of resources on a device. However, compared with a desktop PC, these resources will always be limited.

- *Network Bandwidth Limitations and Instability*

  Wireless networking is the foundation of mobile computing. Without wireless networks, mobility cannot really be achieved. At present, the wireless technologies include Bluetooth, General Packet Radio Service (GPRS), IEEE 802.11a, IEEE 802.11b, IEEE 802.11g and several others under development. The bandwidth available for wireless network is, no doubt, limited when compared to wired networks. At present, the largest possible bandwidth (54Mbps) is provided by IEEE 802.11a. However, many devices could be using 1-2Mbps of Bluetooth connections, or a 114 kbps GPRS connection. Also, the frequency spectrum in which communication is carried out is shared. This could lead to interference in communication. In addition, connectivity is affected by network coverage. Some areas may have very weak coverage or no coverage at all. In short, we have a limited bandwidth network affected by interference, random range of coverage and network failures, leading to a very unstable network environment.

- *High Mobility*

  Users are provided with unrestricted mobility and connectivity, encompassing both user and device mobility. They can carry out their task while moving from one location to the other, as well as switching from one device to another. For example, a user is carrying out a video conference on his way home. He may be using the digital display in his car for the conference. While walking from his car to his home, he switches the conference to his PDA. Once he is at home, in his room, he continues the video conference on his personal

computer. This simple scenario exemplifies the highly mobile characteristic of the computing environment.

- *Context Awareness*

  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. Let's consider the two examples discussed earlier. In the coffee shop example, the application is able to discover nearby services, and take advantage of them, such as the printing service offered by printer in the coffee shop, enabling the user to print his document. Applications are able to adapt to the input and output capabilities of the device being used. In the video conferencing scenario, the output adapts from the digital display of the car, to perhaps a monochrome display of the PDA, and finally to the full-fledged color display of the PC.

- *Proximity Interactions*

  Computing is not longer carried out in an isolated manner in which the only interactions are those with servers. In fact, a major part of the interactions will be carried out with devices which are in close vicinity – forming peer to peer networks for co-operation, accessing services, sharing resources and information. Continuing with the video conferencing scenario, when the user walks into his room with his PDA, the PDA automatically detects and joins in the peer-to-peer network already existing in the room. It discovers the PC in the network, and negotiates for the transfer of the application. It would need to share information in order to ensure a smooth transition of the application.

2.1.4 Infrastructural Requirements

Current computing infrastructure and architectural models are inadequate to provide for the new features of mobile computing. It calls for a change in software systems, network systems, in applications and, in facet, in the whole architecture. Software systems rely on the underlying architecture for support. Therefore it must provide certain basic services in order to realize a truly seamless mobile environment. Some of these requirements are discussed below.

- *Location Tracking*

  Users are constantly moving. The system must be able to determine the location of users. Some systems utilize special hardware to detect the location, such as an infrared

transceiver system [87], while others may use a commercially available Global Positioning System (GPS), which works only outdoors. Castro et al. [10] propose a method to infer the location of a client in an indoor wireless LAN environment from signal quality measures. Once the location of a device is determined, this information is usually stored on location servers present in the network where it can be queried.

- *Data Delivery and Synchronization*

  Users access their data from various locations and devices. The changes they make on the data from one device must be reflected when they change their device or location. In other words, they must be presented with a consistent view of their data. Work is being carried out on distributed information storage and retrieval systems, such as, Chord [79] and Freenet [16]. These utilize mapping algorithms - given an identifier, they will determine the node responsible for storing that identifier's value, which could be a data value or a document. Coda [74], a file system for mobile clients, supports disconnected operations and utilizes operations-based update propagation to save network traffic. In addtion, frequently accessed data is often cached on the device. Disconnection and client mobility make maintaining cache consistency a problem [7]. The industry also recognizes the need for data consistency and have come up with SyncML [81], which is an open industry standard based on XML, for universal synchronization of remote data and personal information across multiple networks, platforms and devices.

- *Mobile Networking*

  Mobile hosts will, no doubt, be using wireless channels to connect to the Internet. The network topology can be constantly changing as the device moves from cell to cell. Moreover, the radio frequency spectrum, which is used for communication, will probably be shared with the other devices in the vicinity. A lot of research is being carried out to deal with the network access issues of mobile computing which includes research in protocols, packet scheduling, channel allocation, admission control and bandwidth management [4, 37, 46, 84]. For example, MobileIP [41] maintains addressing of mobile nodes while maintaining compatibility with existing Internet protocols

- *Ad-hoc Networking*

  Devices communicate with those in close proximity, usually by forming ad-hoc networks among each other. An ad-hoc network is a collection of wireless mobile hosts, forming a temporary network without the aid of any established infrastructure or centralized administration [45]. In such an environment, due to range limits or other factors, one device may need to seek the help of another in forwarding a packet to its destination.

Royer and Toh [73] provide a review of current routing algorithms for ad-hoc networking in wireless environments. The various algorithms can be classified under two categories: table-driven and source-initiated algorithms. Table-driven algorithms require each node to maintain one or more tables storing routing information, thereby always maintaining a consistent network view. Source-initiated routing algorithms create routes only when desired by the source node, i.e. they probe a route only when a message needs to be sent. Most of the time, devices rely on information from neighboring hosts, but some systems also use geographical positions to help routing decisions [55]. However, geographic position based routing schemes have very limited use in indoor networks with closely located nodes.

- *Context Determination and Dissemination*

  Mobile applications, many times, depend on the context in which they are running. They rely on, (1) locally available data, such as memory, the available bandwidth, and also on (2) information regarding the surrounding environment such as printing services available nearby. Determination of some context information needs infrastructural support, such as locating nearby objects and services and the end-to-end bandwidth available. Huang et al. [39] argue that information is, many times, not useful at the time and place it is generated. Rather, it must be re-presented later where it can be acted upon. In that sense, the infrastructure will have a major role in the timely and relevant delivery of context information. Research on context determination and dissemination is still in its early stages [13]. Most research focus on location determination, network bandwidth availability, nearby service discovery and device resource usage [13, 35]. As the field becomes more mature, we shall see determination of a lot more contextual parameters, and a more integrated infrastructural support for context delivery.

- *Support for Adaptation*

  As one can see from above, what makes mobile computing difficult is the inherent characteristic of variation and incessant change. Everything - device resources, network conditions and contexts, are constantly changing. Any application, middleware or system targeted to such an environment needs to be able to detect and dynamically adapt to these changes. Applications will rely on infrastructure for adaptation. In some cases, the infrastructure itself needs to adapt some of its components, for example routing mechanisms, transmission protocols, etc. [37, 45]. More on this will be discussed in the next section.

2.2 THE NEED FOR FUNCTIONALITY ADAPTATION

2.2.1 Characterized by Variation and Change

The mobile computing environment is characterized by *variation*. It is marked by the use of a wide range of devices and technologies. There is no universal device, network characteristic or configuration. In addition to this variation, environments are *constantly afflicted by change* – a device may run out of power, or use a different device, or move to a location with lower bandwidth. Thus, applications and systems targeted for such a dynamic environment should cater for the incessant variation. They have to be able to detect run-time changes and adapt to them appropriately.

Such variation and change can be broadly classified along the following axes:

- *Device and Run-time Resources*

  There is a huge plethora of devices which can be used for mobile computing and that range of devices will keep on widening. As mentioned in the previous sections, these devices have different processing powers, memory sizes, display capabilities, input devices, output devices, etc. Applications need to cater for this variation in device configuration. More importantly, the amount of resources keeps changing at run-time, such as available memory and energy. With the limited amount of resources these devices have, any change would have a pronounced effect. Application and systems must gracefully adapt to the change.

- *Network Infrastructure*

  Devices connect to the network through various networking technologies - wireless LANs, Bluetooth and GPRS, each with different bandwidths, access protocols, and error rates. In addition, as mobile hosts move from one location to another, network characteristics change. Bandwidths may change if the host moves into an area with low network coverage. A device may also move into an area of whose network it has no prior knowledge. It will, no doubt, have to learn at run-time.

- *User Preferences and Environmental Factors*

  Not only do the device and network affect applications, they also need to take into account user preferences and environmental considerations to provide appropriate,

intuitive and customized services to the user. These considerations include time, location, nearby devices, nearby people, etc.

Often, when reading literature on mobile computing, one comes across the terms "context" and "context-awareness". Any factor which affects an application's behavior can be considered as part of the context. Chen and Kotz [13] have divided contexts into 4 categories

- *Computing context*, such as network connectivity, communication bandwidth, nearby resources, etc.
- *User context,* such as user profiles, location, social situation, etc.
- *Physical context,* such as lightning, noise conditions, temperature, etc.
- *Time context,* such as time of a day, week, month, season, etc.

In other words, mobile computing can be seen as being affected by a constant change in context. Applications and systems need to adapt to contextual changes i.e. they need to be context-aware. For the clarity of our discussion, in this chapter, we categorize these contextual changes into four types

- *Device Resources.* These include factors internal to the device, such as the working memory available, the processing power, the energy, etc
- *Network Properties.* These are changes in the network characteristics, such as the network bandwidth, network type, protocol, etc.
- *Environmental Context.* This includes factors in the surrounding environment, such as location, entities available nearby, time, etc.
- *User Preferences.* These are specific choices which the user has made to decide the execution of a particular application.

2.2.2 Adaptation and Adaptability

Adaptation is fundamental to mobile computing. It is currently the focus of extensive research. Some of the techniques being explored include data adaptation, energy-aware adaptation and rate adaptation. The word adaptation is so widely used that it has come to mean two different concepts, which are discussed below.

We want applications to be able to change at run-time, say, to adapt the amount of memory available they use, the network bandwidth they take up, or maybe, when running out of power, switching to low-power mode. They should be able to detect changes in the

environment and respond to them appropriately. This is the "end result" we want applications to achieve.

On the other hand, applications can employ various ways to respond to the change – "the means". By changing the quality of the data accessed, say, a poor quality image, both network bandwidth and memory can be saved. In other words, you are adapting the data quality, in order to achieve two "ends".

There is a slight, but significant, difference between the two concepts: one is the end and one is the means. Most literature use the term "adaptation" to mean both (i.e. using data quality adaptation to achieve network adaptation and memory adaptation). However, for the sake of clarity of the discussion, we differentiate between them.

- *Adaptability – the ability to change* a run-time characteristic in response to a trigger. It is the "end" we want to achieve. If a piece of software supports network adaptability, that means that it can adapt its network behavior according to different network environments.

- *Adaptation – the action taken to achieve adaptability* by change of a certain property, parameter or metric. It is the "means". If an application changes the quality of the data it accesses according to the network characteristic, it is employing data adaptability.

Hence, adaptation is used to achieve adaptability. For the example above, we can say that we use data adaptation to achieve memory and network adaptability.

Adaptive applications and systems have adaptability. They can change their run-time characteristics in response to an external trigger or change. These run-time characteristics can be considered in several dimensions which include:

- *Memory adaptability.* The ability to adapt to the changes in run-time memory available. Every device has different amounts of memory and processing power available. An application should have the ability to provide the same functionality in a resource-rich PC and in a resource-constrained PDA environment. Moreover, with several applications running at the same time, it is possible that one application may find itself suddenly out of memory. In that situation, it should gracefully adapt to perhaps a more memory efficient mode.

- *Energy adaptability*. The ability of an application, or system, to adapt its energy usage. Power is one of the most limited resources in a mobile environment. Applications and systems should be able to run in a more energy efficient manner, in situations of limited power supply.

- *Network adaptability*. The ability of systems to change their network behaviors in response to changes in the network infrastructure. They should be able to reduce their bandwidth requirements, accept a greater degree of packet loss or be able to connect to a new network environment as the need arises.

- *Device adaptability*. The ability to adapt to device configurations. Users move from one device to another. Applications will need to follow the users. The devices may have different input and output capabilities. For example, a PDA may use a pen-based input, whereas a laptop uses a keyboard. They may have different processing powers. It involves more than just changing the device drivers. The presentation format, the UI and perhaps the application logic also need to adapt to the new configuration. Applications need to provide a seamless transition from one device to another.

- *Context adaptability*. The ability to adapt to various external factors. This encompasses everything else that can affect an application, including location, time, user preference, presence of nearby entities, etc.

There are many techniques, both hardware and software, which are currently used to achieve the various forms of adaptability. Many times, change in one metric can affect two or more dimensions of adaptability. As seen from above, by being able to change the quality of data accessed through the network, one can achieve both network and memory adaptability.

Adaptability embodies two mechanisms. One, to detect the changes in the factors of interest, such as the network environment and the amount of resources, etc. Usually there is an entity, such as a resource manager or a network manager, which detects the change. This entity can be located in the mobile host, in the network, or even on servers. The second mechanism is to respond appropriately to the change. A lot of factors come into play when deciding how to respond, such as what metric to change and where to carry out the change. Often, you have to take into account several conflicting criteria. For example, when sending data over a network, using a higher bit rate may drain more power resources than sending the same piece of data using a lower bit rate. Tradeoffs need to be made.

Very often, some sort of adaptation is employed as a response to the change. Adaptation can be seen as a means of achieving adaptability. According to our definition discussed above, adaptation implies change of a certain metric or parameter. Adaptation changes a certain parameter and the effect of that change leads to adaptability. We can roughly categorize adaptation into five categories according to the metric or the factor changed.

- *Data Adaptation*

  Mobile applications usually need to access data for information or for entertainment, such as emails, stock quotes, web pages, multi-media, etc. Data adaptation involves *changing the data in some manner*, such as changing the quality of the data accessed, transforming data to a more appropriate form, accessing a different set of data altogether, etc. This is the basis of many of the transcoding and content adaptation techniques. There are several projects which use proxy-based data adaptation to change the quality, or *fidelity*, of the data accessed, on the fly according to the client resource available, or according to the network environment [24, 12, 34]. For example, in Odyssey [23], the server has several pre-generated versions of the data with different fidelity levels, and the appropriate one is chosen at run-time according to the resources, or even energy, available.

- *Network Level Adaptation*

  This involves the change of network level parameters. This can be done in many ways, for example, by using different protocols, adapting routing information, changing the rate of transmission, network level QoS management, etc. There is a lot of research in this area covering a wide spectrum of adaptation techniques, which lead to network adaptability and even energy adaptability. A few examples include dynamically changing data rates to match channel conditions [37], deploying filters to an intermediate proxy to filter or delay all but the most essential data [91], using an adaptive communication protocol for energy conservation [36], changing routing information according to location of the client device [41] or even according to the energy available in the nodes [54].

- *Energy Adaptation*

  This involves using techniques which change the amount of energy consumed, either by turning the power off or moving to an less energy-consuming state. Many of techniques are hardware related, such as dynamic voltage scaling of the CPU [67], adapting hard disk spin policies, etc. Simunic et al [77] have devised a dynamic power management policy which can be used on small devices, laptops and wireless LAN cards. Xu et al. [88],

carry out power adaptation in ad-hoc wireless networks by turning off unnecessary nodes without affect the routing capability.

- *Migration Adaptation*

  This involves changing the location of execution, for example, by moving to another machine with more resources. This is often used in the fields of distributed computing and mobile agents, whereas in the field of mobile computing, it is still rare or under development. Adaptive distributed applications and mobile agents migrate to nodes, which fulfill the resource requirements, if they realize that the current node does not have sufficient resources for their execution [18,33]. Migration adaptation can be advantageous in a mobile environment, for example, migrating from a PDA to a nearby laptop in order to speedup execution.

- *Functionality Adaptation*

  This involves changing the way an application carries out its functionality. Applications in a mobile environment can be seen as fulfilling certain tasks [5]. Functionality adaptation implies carrying out the same task but in a different manner, either by using a different mechanism, different algorithm, a different QoS characteristic, or by switching to another execution mode, etc. It involves changing the execution of the task. For example, if a device does not have sufficient computation power, an application can use a smaller key for encryption. More details on functionality adaptation will be discussed in Section 2.2.3.

There may be techniques used to achieve adaptability which do not fall into any of the above categories. As time goes by and more techniques are discovered and employed, the categorization has to be broadened and refined.

Notice that we have not included context adaptation as one of the adaptation techniques. As mentioned earlier, adaptation implies change of a certain parameter. Hence, context adapta*tion* would mean that we change the external contexts in a certain way. Up to the current moment, to the best of our knowledge, such a technique has not been applied. Context adapta*bility*, on the other hand, means *responding* to contextual changes.

2.2.3 Functionality Adaptation

Functionality adaptation is probably the most versatile and the least explored of the five adaptation categories. In a mobile environment, user interaction can be seen as users performing certain tasks [5]. Thus, applications provide a means to carry out a group of tasks, i.e. an application provides a set of functionalities which a user can execute to fulfill his task. Functionality adaptation involves changing the way the task is carried out in order to respond to the changes in the mobile execution environment or context.

The other adaptation techniques involve changing lower level factors, such as the network transmission parameters, data quality or the power utilization. They do not really change how the assigned task is executed. Functionality adaptation involves *changing the execution* of the application. It is a software level adaptation technique and some of the ways it can be carried out include:

- using different sequence of actions or different algorithms,
- using code enhanced for certain platforms,
- changing the memory usage and processing time balance (i.e. using less memory but more processing time to carry out a task), or
- partitioning the task so that it is partially executed on a server rather than completely locally.

Let's consider a simple example of an email application. When checking mail, the application downloads all the new messages, including the body and attachments if it is running in a device which has sufficient resources, such as a laptop. However, if it is running on a resource-limited, network-constrained device, such as a mobile phone, it downloads only the headers of the messages - the sender and the subject fields. When the user wants to read a message, only then does it download the body of the message. To a user, this difference in the actions taken is transparent. The application achieves functionality adaptation by using different sequences of actions to fulfill its task or, in other words, its functionality.

At a lower level, it can be said that functionality adaptation involves using different chunks of code for execution under different environments. Mechanisms for achieving this include, among others:

- using different parts of the application code, i.e. different subroutines, or modules in different scenarios. Which part of the code is used is decided by the application itself, or by the underlying middleware.

- using completely different application code i.e. dynamically replacing a module or a chunk by a more appropriate one. This requires that the software architecture and runtime infrastructure support replacement.
- delegating the execution of the code to some other entity (this could involve code migration). There could be active delegation, i.e. the application code decides which part to delegate and to whom. Or, it could involve passive delegation – the underlying system decides what and where to delegate to.

Functionality adaptation is, in fact, a very powerful tool. If implemented appropriately, it can be a comprehensive technique which has the potential for enhancing an application's or a system's adaptability to a very wide range of factors, including network bandwidth, device resource availability, and environmental context. For example, in a network-constrained environment, it can delay the download of the body of an email until the message is actually accessed. Depending on the device display capabilities and memory available, different codes are used for viewing email.

Unfortunately, research in functionality adaptation in the context of mobile computing is rather limited and the mechanisms used are rather restricted. In Odyssey [23, 62], for example, the system notifies the application when there is a change in the resource of interest. The application responds by accessing a different fidelity of data, which may need a different decoding mechanism altogether. Functionality adaptation, in this case, involves adapting the application so that it can process different levels of data fidelity. Data adaptation and functionality adaptation go hand-in-hand in this case. You cannot change the data fidelity if the targeted application does not have the appropriate code to process it.

Another example is the architecture proposed by Kunz and Black [51]. In a client-proxy-server scenario, the application logic is dynamically split between the mobile client and the proxy in order to adapt to the dynamic wireless environment. This is referred to as *application apportioning*. Applications register certain information with the run-time system about when to carry out partitioning, for example, execute a certain code on the proxy if the bandwidth is smaller than a value. At the appropriate time, the application logic is moved by using object migration mechanisms.

The main difficulty with the above approaches is that the adaptation policy is ingrained in the application code. The programmer has to decide the adaptation policy at design time and hence, it cannot be dynamically extended. This leads to three major drawbacks:

- *Burden on Application Programmer*

  The programmer has to decide when to adapt and how to adapt. He has to incorporate the different versions of the same functionality into the application. Not only that, he has to determine the adaptation policy at design time, i.e. when to use which version of the functionality. This places a burden on the programmer, who has to design for all possible scenarios.

- *Bigger Application Size*

  Since all the different functionality versions are packed into the application, an adaptive application has a bigger size when compared to a non-adaptive one. With memory being a limited resource in a mobile device, such a feature goes against adaptive programs.

- *Limited Adaptive Capability*

  The different adaptation versions are determined at design time. As mobile environments evolve, incorporating new types of devices, technologies, etc, the adaptive nature of such applications become outdated. The adaptation cannot be dynamically changed, or extended. For every new range of devices or technologies introduced, the application may have to be rewritten or reinstalled. This is not feasible in a fast-paced arena such as mobile computing.

Functionality adaptation, no doubt, is essential and important in a mobile computing environment. A mechanism for achieving functionality adaptation needs to be developed which can overcome the above problems. Such a mechanism

- should not increase the application size considerably,
- it should be able to incorporate features of new technologies without having to go through the trouble of rewriting or reinstalling the whole application and
- it should reduce the burden on the application programmer.

You may notice a similarity between functionality adaptation and dynamically reconfigurable software systems [48, 49, 75]. The difference lies in their purpose and targeted execution environment. Functionality adaptation occurs in programs targeting the mobile computing environment. "Reconfiguration" is carried out in response to changes in the run-time environment or context. The main purpose for reconfiguration is adaptation. On the other hand, configurable systems usually are long running server-side applications. "Reconfiguration", in this case, is carried out for software upgrading, that is, for software extension. Methods for achieving functionality adaptation may be able to learn from some of the techniques used in dynamic reconfiguration.

2.3 SUMMARY

In this chapter, we see that mobile computing is an emerging trend. Mobile computing can be seen as the use of mobile devices, taking advantage of wireless means, to create a seamless computing environment, enabling access of information, computation and co-operation. Mobile computing is different from pervasive computing, in the sense that it puts its focus on mobility rather than on filling the living environment with smart devices.

What sets the mobile computing environment apart from other environments is its device heterogeneity, device resource limitations, network limitations and instability, high mobility, need for context-awareness, and the proliferation of proximity interactions. To realize the ultimate mobile environment, infrastructural support for location tracking, data delivery and synchronization, mobile networking, ad-hoc networking, context determination and dissemination, and adaptation are needed.

The difficulty of mobile computing lies in its highly dynamic environment. Every aspect is affected by change and variation, including resources, networks properties and context. Systems and applications need to take into account these changes and respond to them accordingly. In this chapter, we have differentiated between adaptation and adaptability.

- *Adaptability* is the ability to change a runtime characteristic in response to a trigger. It is the ability to detect a change in the external environment and respond to that change.
- Adaptation is the action taken to achieve adaptability by change of a certain property, parameter or metric. The change of a single property often affects more than one run-time characteristic.

We looked at several types of adaptability including resource, energy, network, device and context adaptability. We also explored the different adaptation techniques currently employed to achieve adaptability including data adaptation, network level adaptation, energy adaptation, migration adaptation and functionality adaptation.

Out of the five adaptation techniques, functionality adaptation provides a comprehensive mechanism which has the most impact on the adaptability of a software. However, current research in functionality adaptation, in the context of mobile computing, is limited and is lacking. The major drawback of some of these techniques is that they put a lot of burden on

the programmer who has to decide the adaptation policy at design time and hence, cannot be dynamically extended. There is a need of a functionality adaptation technique which can overcome the drawbacks.

Chapter 3

Component-Based Technology

Component-based software development has received interest from both the commercial and academic sectors in recent years. Components allow the commoditization of software. Software systems can be built by integrating already existing software components rather than from scratch. From the development point of view, software components enable greater code reuse, reduce development time, enhance maintainability and thus reduce development costs.

Often an analogy between software components and hardware components in integrated circuits is brought up. Just as how a modem can be made by assembling different IC components and wiring them together, similarly, software systems can be built by wiring together different software components. Software components, hence, have become commercial-off the shelf (COTS) products [9, 82].

This chapter provides a brief overview of current component-based technology. We first define what a component is. Then we describe the stages involved in carrying out component-based development. The rest of the chapter looks into current leading component models including CORBA, JavaBeans, Enterprise JavaBeans and .Net Assemblies.

3.1 WHAT IS A COMPONENT?

Before we can continue our discussion, we must first define what a component exactly is. At present there are many similar but no single definition of component. For example, Szyperski provides the following definition of a component in his book [82].

"*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"

Similary, D'Souza and Wills [21] define a component as follows.

"A component is a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system."

Even though the definitions are different, they agree that a component has the following three characteristics.

- *A component is independent*. Independence does not mean that a component does not depend on other components. It just implies that the dependencies are general enough for several different providers to satisfy [6].

- *A component provides functionality via well-defined interfaces*. Each component can have multiple interfaces, each representing a service the component offers. These interfaces are of contractual nature with well-specified input and outputs, so that the services of the component can be used. They emphasize the black-box nature of a component.

- *Components can be used for composition.* This is the most important characteristic. For components to be composable, they must come with clear specifications of what they require and what they provide. They must specify what the deployment environment will need to provide for the execution of the component, for example, the required interfaces of other components i.e. the dependencies. They also must specify a complete listing of the services provided including any error conditions that may occur.

Another characteristic of a component, which is often implied, is that *a component has no persistent state* [82]. It cannot be distinguished from copies of its own. A component can be activated and loaded into a system. Different instances of a component may be available at the run-time, however there is only a single copy of the component in a system.

3.2 COMPONENT-BASED DEVELOPMENT

As mentioned earlier, components can be composed to create software systems. Granularity of the components is determined by the programmer. Components connect to each other by invoking one another's interfaces. Interfaces are the means by which components connect. A component provides an interface to a functionality, and the client calls that interface to access the functionality.

Interface specifications can be considered as contracts between clients of the interface and implementations of the interface. They become the mediating link between providers and clients which may be ignorant of each other. The contracts usually include the syntactic definition of the interface, the pre-conditions and the post-conditions of the operation – the functional requirements. Non-functional requirements such as the performance, the resource requirements, etc, should theoretically be part of the contract. However in many commonly-used component models, they are often left out.

Component-based development can be divided into four phases [8, 74].

- *Component Qualification* – determining which existing components are fit for use in the new system context, keeping in mind both functional and the non-functional requirements.
- *Component Adaptation* – fine-tuning the component so that it fulfills the requirements of the new system. The degree to which a component can be adapted depends on the individual component.
- *Assembling Components* – integrating the components so as to form a software system from the disparate components.
- *System Evolution* – adding, removing or replacing a component so as to upgrade the system, add new functionality, etc.

One must keep in mind the difference between the run-time and the static characteristics of software. Component-based development, no doubt, creates a modularized static software structure. The software code is divided into components which are connected to form the system. However, the run-time characteristic may be very different. In some systems, the component boundaries are indistinguishable at run-time, or even load-time, hence creating a big monolithic chunk of software. On the other end of the spectrum, some systems enable components to be looked up and hooked at run-time, creating a dynamic component infrastructure, allowing components to be replaced at run-time. Dynamic component

composition falls under the latter category of component systems. Most systems fall in between the two extremes. Software is built from components. All the constituent components are packaged and distributed as part of the application, even though they may be linked at run-time. From the application user's point of view, software is still distributed as a monolithic chunk. For dynamic component composition, however, software is distributed component by component.

The benefits of component-based technology include reusability, understandability, reduced development costs and dynamic extensibility. However, the feature, which is of most importance to mobile computing, is its adaptability. Component systems possess the capability to change configurations by adding, replacing or removing the constituent components, adapting the software as desired.

3.3 CURRENT COMPONENT TECHNOLOGIES

In this section, we look at some of the current component technologies, namely COM [17], CORBA [64], JavaBeans [78], Enterprise JavaBeans [72], and .Net Assemblies [59]. Most of the components cannot run on their own and need to execute inside another process which is often called a container. Some, however, can run by themselves and provide mechanisms for other processes to access their services. Not all components need to be located locally at the client. Some components may be located on servers. Technologies such as DCOM [17] and CORBA, provide the wiring required to connect up distributed components.

Essentially, most of the commonly used components can be divided into four categories [74] which are:-

- *In-process client components.* These components run inside containers, or applications, such as ActiveX, which are based on COM, JavaBeans, and .NET Assemblies
- *Standalone client components.* These components can run on their own, exposing their services to other programs through interfaces, for example, OLE Automation (which is again based on COM) and .NET Assemblies.
- *Standalone server components.* These components run on a server machine, which can be accessed through remote procedure calls or other networking communication. Leading technologies include DCOM and CORBA

- *In-process server components.* These run inside containers such as transaction servers on the server machine. Leading technologies include MTS [58] and Enterprise JavaBeans

In many the component models, the terms "component" and "object" are used interchangeably. It is true that components can be implemented by object-oriented technology. However, the terms are not equivalent. A component is a unit of composition and does not have persistent state. An object, on the other hand, is a unit of instantiation and has a state that can be persistent. A component may be realized by traditional procedures, assembly language or by using objects. A component may contain multiple classes, but a class is definitely confined to a single component. The main difference lies in the roles of components and objects. The role of components is to capture the static properties of an architecture whereas the role of objects is to capture the dynamic nature of systems built out of components [83]. A component can be considered as a static entity, while an object is a dynamic entity, only accessible at run-time. In the discussion below, we have tried as much as possible not to interchange the two terms.

3.3.1 Microsoft's Component Object Model (COM)

Microsoft's Component Object Model (COM) is a software architecture that allows applications to be built from *binary* software components. COM allows components to interact as long as they stick to the binary standard specified by Microsoft. Distributed COM (DCOM) is an extension to COM which allows network-based component interaction. COM and DCOM, in fact, lay the foundation of many other Microsoft technologies such as object linking and embedding (OLE), dynamic link libraries (DLL), ActiveX, COM+ and MTS.

A COM component is an executable block of machine code which implements a set of interfaces through which clients can access its services. Interfaces form the only points of contact between the clients and the component. Every component implements an IUnknown interface which is used to gain access to other interfaces of the component. Interfaces are specified using Microsoft Interface Definition Language (MIDL). Once the interface is defined, it cannot be changed, i.e. interfaces are immutable.

Both interfaces and components are given globally unique identifiers, Interface ID (IID) and Class ID(CLSID) respectively. A client dynamically locates components by querying the registry with the class identifier, CLSID, of the required component. A new instance of the

component is created and returned to the client. The client will then query the instance to acquire a reference to the required interface, through which it can access the functionality it needs.

An important aspect in COM is that, instances of components have no identity. A client can request for a component of a particular type (i.e. CSLID), but not a particular instance. Every time a client requests for a component, a new instance is returned.

Components run either in the same process as the client, or in a different process which may be on the local computer or in another computer across the network. In the case that it is located in a different process, the client is given a proxy object is through which it can connect to the component. The location of the component is transparent to the client. DCOM handles all the details of marshalling and unmarshalling when the components are located in different computers.

ActiveX controls are essentially COM components which implement special interfaces and may incorporate a user interface. They can only be used with OLE containers, for examples Internet Explorer and Visual Studio, which are aware of ActiveX controls. If an ActiveX control is embedded in a web page, Internet Explorer will download the control and then execute it in the same process. In Visual Studio, they can be used to add functionality to an application visually. This visual property is similar to JavaBeans. Please refer to Section 3.3.3.

3.3.2 OMG's Common Object Request Broker Architecture (CORBA)

CORBA is a distributed object architecture which allows application components to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed [11].

The main abstraction in CORBA is a CORBA object. CORBA objects can be considered as application components which provide services through specific interfaces. Interfaces serve as contracts between clients of the services and the objects.

Firstly, an object's interface is defined in OMG IDL – Interface Definition Language. When the IDL is compiled, it will generate a skeleton and a stub. The skeleton code is compiled with the object's implementation and the stub code is compiled with the client code. Both

clients and objects are installed over an ORB – Object Request Broker. They may be installed on the same machine or on different machines, each with an ORB.

Every CORBA object has an object reference which identifies its instance. A client will access an object through its IDL interface by specifying its object reference. A client obtains object references either by receiving them as output parameters on invocations on other objects for which they have references, such as the Naming and Trading Service, or by de-stringification of a "stringified" object reference. The invocation of the service goes to the local ORB via the IDL stub. The local ORB will route the invocation to the remote ORB, which will pass it on to the object implementation via the skeleton. The location of the object and the details of the routing is completely transparent to the client.

Since the client needs to be statically linked to the stub of the objects, it is inflexible. CORBA provides a Dynamic Invocation Interface which allows clients to discover new objects, interfaces and interact with them at run-time even if they are not linked to their stubs.

In CORBA 3, a new abstraction called the CORBA component was introduced together with the CORBA Component Model (CCM). CORBA components are simply special CORBA objects which have been programmed to a special style of interface. CORBA components are server-side objects which are installed in a CORBA container. The container provides services such as persistence, transactions, security and notification, so that the CORBA component programmer can focus solely on business logic. CORBA components are compatible with Enterprise JavaBeans (see section 3.3.4).

CORBA objects can implement only one IDL interface. A CORBA component can bear more than one interface, each with its own object reference. These multiple references are known as facets. Clients of the component can navigate from one interface to another at run-time. Usually, components are distributed as a .car file which contains an XML component description together with the component executable. However, several components may be packaged together as assemblies (.aar files), which are used as units of distribution.

3.3.3 Sun's JavaBeans

A JavaBean is a reusable software component that can be visually manipulated in builder tools to allow programmers to manipulate functionality in a simple and visual way. JavaBeans exhibit four properties: customizability, introspection, events and persistence.

JavaBeans have *accessor* and *mutator* methods which can be used to customize the properties of a bean. Beans can also look into other beans to discover their variables, methods and events. Interaction between beans usually takes place through events. Beans are usually stateful. Hence provide mechanisms to save and restore their state.

Beans are essentially Java class files which follow certain conventions. They extend the Bean class and are serializable. Access to all properties is provided by methods which begin with get and set. Beans are usually packaged in JAR files and are identified by their class names. Developers use visual tools, such as bean box, to compose the components and make applications or applets. The applications are distributed to the client as a whole, together with all their constituent beans. JavaBeans can be considered as development components rather than deployable components.

3.3.4 Sun's Enterprise JavaBeans

Enterprise JavaBeans (EJB) is an architecture for server-side components making it easier to build middle-tier, server-side business applications. An enterprise bean is a server-side software component, which is made of one or more Java objects, and exposes a single component interface. Enterprise beans are deployed in an EJB container, which in turn is located on an EJB server. The container provides a run-time environment for the component to execute in and provides a set of common services to the beans running in it, such as transaction management, security, resource management and life cycle management, persistence, remote accessibility and location transparency etc.

There are two types of enterprise beans – *session beans* and *entity beans*. Session beans implement the business logic of an application. They live only as long as the lifetime of the calling client. Sessions beans can be stateful. Especially in the case that the implemented business process spans multiple requests, the bean needs to retain the state on behalf of the client. In other cases, session beans are stateless. Entity beans, on the other hand, model permanent data. They are long lasting and can serve multiple clients at one time, unlike session beans for which an instance can only be used by one client.

An enterprise bean exposes a single interface which exposes methods for clients to invoke. When a client tries to use a method in an enterprise bean, the invocation is actually intercepted by an EJB object and then delegated to the bean. The EJB object is a part of the container. It is network aware and acts as a glue between the client and the bean. A developer

will create a remote interface which is specific to a particular bean, and the EJB container will auto-generate the EJB objects. A client needs to obtain a reference to an EJB object before it can use the bean. It achieves that by querying the home object. Home objects are responsible for creating EJB objects, finding existing EJB objects and removing them. Home objects are also part of the container. They are generated by the container from bean-specific home interfaces provided by bean developers. The enterprise bean, the EJB object, and the home object, are located in the container.

In order to use a bean, a client first has to look up the home object of the required bean. This is done by querying the Java Naming and Directory Interface (JNDI) for the nickname of the bean. Then the client uses the home object to create an EJB object in the container on the server. The client then uses the reference to the EJB object to call the methods on it, which of course are delegated to the actual enterprise bean. Once the client is done with the bean, it needs to remove the EJB object, i.e. destroy it from the container. RMI-IIOP or CORBA is used for network communication.

Enterprise beans are distributed as Ejb-jar files. Other than the enterprise bean classes, the remote interface and the home interface, the Ejb-jar also contains an XML deployment descriptor which specifies the component's service requirements, such as transaction, persistence, etc. The Ejb-jar also includes Bean-Specific Properties files which the bean can read at runtime to customize how the bean functions. The Ejb-jar files are then deployed on an EJB server such as the BEA WebLogic server.

3.3.5 .NET Assemblies

.NET Assemblies are the primary building blocks of an application for Microsoft's .NET Framework. An assembly is a collection of functionality that is built, versioned and deployed as a single implementation unit. An assembly can be considered as a DLL or a COM component targeted for .NET Framework's Common Language Runtime (CLR).

The main difference between a .NET assembly and a COM component is that an assembly is self-describing. It contains a manifest, which contains the assembly metadata including the naming and versioning information, dependencies, and type information, in addition to code. Due to this reason, registration, separate IDL files, type libraries or proxy/stubs are not required to access a component. Assemblies are mainly identified by their name. They do not need to register themselves with the operating system, unlike COM components

There are two types of assemblies – private assemblies and shared assemblies. Private assemblies are meant to be only for a single application. They stored in the application folder or subdirectory in which they will be only visible for that application. This makes naming the assembly easy since private assemblies have no specific naming requirements except to be unique to the application. Shared assemblies on the other hand are assemblies that can be shared among several applications. They are given a shared name which follows strict naming conventions since the assemblies must be uniquely identifiable across the entire system. They are installed in a centralized repository called the Global Assembly Cache, which is essentially a folder in the file system.

At run-time, the client indicates the name and the version number of the required assembly, it is looked up from the application directory or from the global assembly cache.

3.3.6 General Discussion

It can be seen from above how the theoretical component model has been realized in different ways with different foci and features. COM, CORBA and .NET Assemblies are language independent, whereas JavaBeans, EJBs, CORBA components and .NET Assemblies can boast platform independence.

For some of the models, the linking of components is rather static. It is programmatically decided which component is to be used. For example, for COM, JavaBeans and EJBs, classnames are used to locate the required components. Assemblies use assembly names as well which are programmatically specified within an application. Thus, an application is bounded to a particular implementation of functionality. It can only use that particular component. This reduces the amount of adaptability an application can have. COM, however, does possess a little more flexibility than JavaBeans and EJBs, since each component can implement more than one interface. A client can move from one interface to another during run-time. For CORBA, components are looked up via their object references. Different objects, which implement the same interface, will have different object references. Clients obtain an object reference from the Naming and Trading Service at run-time.

In the above models, the components are not mobile. An application can access a component which is located locally, or a component located on a remote server. The remote component does not move to the client. The component framework provides the wiring for the client to

access the remote component, such as the ORB in CORBA, DCOM and EJBs. In most cases, a proxy is sent to the client at run-time through which a client can access the remote component's services. However, for CORBA, the stub needs to be compiled into the client application.

Java applets on the other hand, move to the client device when they are needed, and are dynamically linked. They also follow certain syntactic constraints. However, applets are not composable, i.e. you cannot put two applets together to make a larger applet, and hence they do not qualify as components.

As for ActiveX controls, they are essentially COM objects which can be used for composition. However, an ActiveX control can be embedded in a web page and it will be downloaded to the client when the client accesses the web page and executed. In this sense, they seem very similar to our facet model i.e. downloaded when required. However, one of the main difference is that ActiveX controls, even though they are dynamically linked to the application, they are not dynamically composed. When you download an ActiveX control, you download the whole control, as one chunk. As for facets, every constituent facet is downloaded separately. In addition, the facet model allow for adaptation to the client device. For ActiveX, the control used is decided at design time, hence limiting the flexibility.

CORBA and EJB are actually distributed object technologies which can be used to realize a distributed component model. When a client application retrieves a reference to a CORBA object, then client is communicating with a specific object instance. If a client stores some data in that object, it can retrieve back some or all of the data later. This is because every CORBA object has an "identity" which can store some "state". In DCOM, however, objects do not have a state or identity. If a client obtains a reference to a remote object and stores data into that object, later when it attempts to retrieve the same object, it cannot. This highlights the difference between the two approaches to distributed components [3].

It can be seen that for the above discussed component models, the contract between components is very syntactic in nature. It only includes the functional aspects. Components connect to each other through interfaces or component names. Often interface definition languages (IDLs) are used to define the interface of the component. What functionality a component provides is implied by its implementation or by the description supplied by the provider. Non-functional criteria, such as performance, resource requirements, etc, are usually not included in the contract.

3.4 SUMMARY

This chapter provides background on component-based technology. A software component has four characteristics

- It is independent.
- It provides functionality via well-defined interfaces.
- It is a unit of composition.
- It has no persistent state.

Interface specifications can be considered as contracts between clients and providers of a component. Theoretically, interface specifications should include both functional and non-functional requirements, but in practice, they often only include the former.

The dynamic characteristics of component systems lie between two extremes. On one hand, component boundaries are indistinguishable at both run-time and load time. On the other hand, components retain their boundaries all through out, even during execution. Dynamic component composition falls under the latter category. The advantage of component systems is that they possess the capability to change configurations by adding, replacing or removing the constituent components.

We looked at five current component technologies including COM, CORBA, JavaBeans, Enterprise JavaBeans and .NET assemblies. Each of them has a different focus and can boast different advantages. Some of them support only local components, such as JavaBeans and .NET Assemblies. Others provide wiring to remote components as well, such as COM, CORBA and Enterprise JavaBeans. However, components in these models, do not move. Their location, once installed, is fixed.

Chapter 4

The Sparkle Mobile Computing Environment

In the earlier chapters, we saw how mobile computing differs from traditional distributed computing. The most fundamental requirement for software is to be able to adapt to the heterogeneity and the fluctuating conditions of the environment. Current software and software distribution models are found to be lacking in this respect.

In this chapter, we look at the current software distribution model, the *monolithic* approach, and explain why it is not suitable in a mobile computing environment. We then propose an approach, which utilizes dynamic component composition, and discuss its appropriateness to mobile computing. After that, we describe the infrastructure of Sparkle Mobile System which makes use of dynamic component composition to achieve functionality adaptation in a dynamic and flexible way.

We then describe the component model employed in the Sparkle System. The components are called facets, because they resemble facets of a diamond. Numerous small facets put together make up a sparkling diamond. In the same way, many components put together can make up a powerful mobile application. Finally, we look at some research carried out on component models in mobile environments.

4.1 CURRENT SOFTWARE DISTRIBUTION APPROACH

At present, the most common way of obtaining software is by purchasing an installation CD from a software shop, or by downloading application files from the Internet and running the installation program. The whole application is installed in the device and can be invoked for use. This is the *monolithic* approach – applications are distributed as monolithic chunks. You have to install the whole application or nothing at all. You have to install the whole piece of

software, even if you normally use, say, one tenth of the functionality provided in that software.

One of the main drawbacks of the monolithic approach is that it puts a limit on the amount of functionality that can be placed on a mobile device. Applications provide users with functions. The more functions an application has, the bigger its size or memory requirements will be. In mobile environments, where devices with limited resources are prevalent, normal applications are too big to be installed. Instead, programmers need to provide smaller, scaled-down versions of the applications with less functionality, in order to fit into the resource constraints of the targeted device. In other words, to fit programs into devices, developers build applications with smaller sizes, which, in turn, limit the functionality provided by the applications.

In addition, applications are usually tailored for a specific platform. It is not possible, though, to write different versions for all the possible configurations in a mobile environment. Moreover, once an application is made for a certain device, it runs as it is designed. Consequently, the applications are designed for the most limited configurations of the target device, for example, the least memory available. It usually is not able to take advantage of the extra memory available if it is running in a device with more memory.

Some applications are adaptive. However, the adaptability is usually hard-coded into the application, which can only accommodate a limited set of configurations, or devices. To be able to adapt to another new configuration or platform, developers need to rewrite the application again, and redistribute it. Such an approach will not be able to cope with the ever-increasing range of heterogeneity of mobile devices.

Also, once an application is made and installed, there is limited support for extending the functionality offered by it, unless you reinstall the application. The exception to this are plug-ins. But, plug-ins provide limited functionality extension. There is a tendency of application developers to bundle an overly rich set of functionality in a single piece of software. Again, the more functions an application has, the bigger the size, the less suitable it is for small devices. Microsoft Word is a good example. It offers a lot of functions to users other than just text editing, such as drawing graphs, editing formulas, macros. The functionality offered increases with every new version of the software. However, most users only use a subset of the functionality, but developers have to include all the functionality together for the sake of completeness. Thus, applications have an unnecessary bulk to them.

In fact, what would be more suitable in a mobile context is being able to extend the application after installation. Applications are written so that they can be installed with only the basic functionality, and are extended according to the user's usage requirements. Basically, applications only contain functionalities which users require rather than including all the unnecessary ones, thereby reducing the size of the application and making it suitable to be installed on small devices.

4.2 DYNAMIC COMPONENT COMPOSITION

As seen from above, current software distribution mechanisms fall short when applied in a mobile computing context. The monolithic approach puts a limit in the amount of functionality which can be placed in a device and cannot flexibly adapt to different device configurations. Thus, a flexible way to distribute software, which overcomes the shortcomings of the current model, is needed.

Our approach utilizes dynamic component composition. Instead of distributing an application as one big monolithic chunk, an application is broken up into small components. A device is installed with a minimal set of initializing components. When the application is run, as components are needed, they are brought in from the network, linked to the run-time system and executed. These components can be brought in from dedicated servers or from near-by peers. Once the components are used, they are unlinked from the run-time system and thrown away. The application is, thus, dynamically composed from components at run-time.

The basic philosophy of this approach can be summarized as the *get-use-throw* approach – you get a component when needed, use it and then throw it when it is done. If you need it again, you get it again. Even though the philosophy may seem simple, it has several advantages over the other approaches:

- *The functionality a device can provide is not limited by its configuration*
  Components are brought in when they are needed and discarded after use. These components are small when compared to full-fledged applications. Being able to discard a component is of utmost importance for computing in small devices. Whatever is unwanted, can be thrown away, freeing up resources and memory for components, which are currently running, or to bring in other components. This enables devices to run programs which, if distributed as one whole chunk, would not

fit in the device. A device can run a program with more functionalities without worrying about the size of the application.

- *Extensive support for functionality adaptation*

  Since components are brought in at run-time, the application can dynamically adapt to the device's run-time environment. For example, there are two components which carry out the same thing, each with different run-time characteristics. The one which is most suitable for the run-time conditions of the device is brought in and executed. In a memory-constrained environment, a component which uses less memory, but perhaps requires more computation may be used. In addition, since the application is "assembled" at run-time, this makes it dynamically adaptable. Applications can easily adapt to new requirements and configurations. All it takes is updating the corresponding components and bringing in the newer components instead of the older ones.

- *Increased scope for peer-to-peer co-operation*

  Many of the current peer-to-peer technologies focus on data and file sharing, such as Napster [60], Gnutella [28]. Instead of sharing only data, clients can share components, essentially forming a component pool. Clients can get components from nearby peers, instead of getting them from far-away servers. For instance, if several users are having a meeting and they all need to access a certain application on their PDAs, they can get the components from each other. In addition, if one of them does not have enough resources to run a component, it can delegate the execution of that component to the peer.

- *Support for user mobility*

  All functionality is brought in from the network at run-time and it adapts dynamically to the client device being used. Hence, applications are not restricted to a particular device or even to a particular location. Users can access their applications from any device anywhere. When a user moves from one device to another, the same functionalities can be brought in, with components suitable for the new device.

- *Enhanced migration adaptation*

  If a client device does not have enough resources to run a component, it can delegate the execution of that component to a peer or a server, as a last resort. Instead of migrating the whole application, it only needs to migrate the execution of one component. In a mobile environment, it is probably more suitable to delegate a single

component rather than the whole application. Not only that, the delegated device uses components which is more suitable for its run-time conditions rather than the original components. This basically enhances migration adaptation with functionality adaptation.

4.3 THE SPARKLE MOBILE COMPUTING ENVIRONMENT

The Sparkle Project aims to provide an Internet-enabled infrastructure for mobile computing which supports dynamic component composition. Applications can be seen as a means by which users perform tasks [5]. An application usually provides several functionalities which users can invoke to fulfill their tasks. Applications can be broken up into components along the lines of functionalities. Every component provides certain functionality. There may be more than one component which fulfills the same functionality. As mentioned earlier, at run-time, the appropriate component is brought in and executed. Hence, it can be seen that *applications are linked by functionalities, rather than by exact components*. Consequently, when you run an application in different environments, to carry out the same task, the actual components used may be different.

In our model, clients send requests for the components to the network, and are returned with the appropriate component. Since applications are linked by functionality rather than by exact components, the requests specify functionality requirements rather than component identifiers. They also include non-functional requirements such as run-time resource information and context, so as to determine which component would be most suitable for the client. From the client's viewpoint, there is a lot of reliance on the network. The network stores the components and also possesses intelligence to match the appropriate component for the client.

Therefore, the network architecture plays an important role in this model. In this section, we provide an overview of the whole architecture, the network entities involved and their roles in achieving dynamic component composition. Components in our system are called *facets*[1].

[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 with other facets, they can create a very powerful application. That's why the project was named "Sparkle", to symbolize a sparkling diamond made of facets.

Every component has a manifest which describes what functionality it fulfills and its run-time behavior, such as the amount of memory it requires or the network bandwidth, etc.

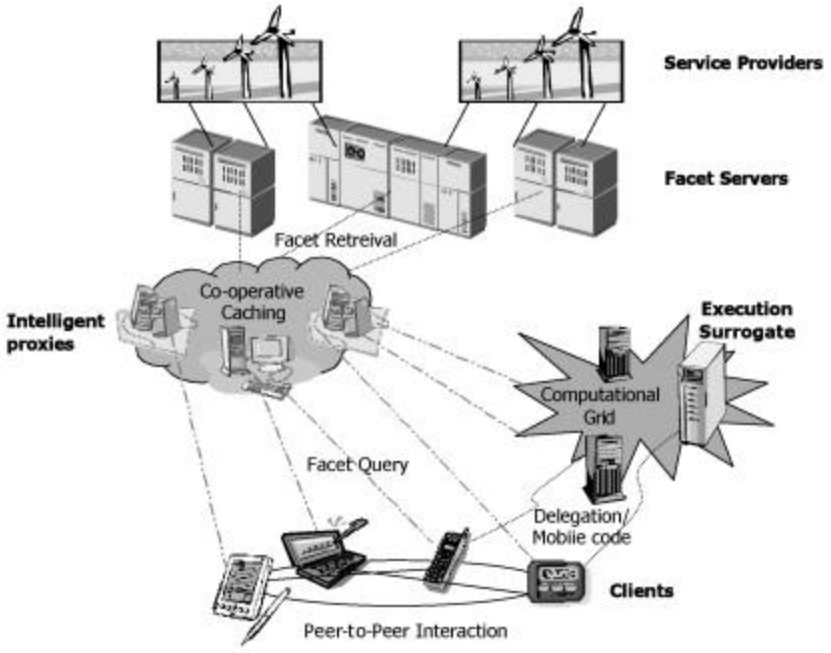


Figure 4.1 Overview of the Sparkle Architecture

4.3.1 Overview

The sparkle architecture consists of several entities which are listed below.

- *Client Devices*

  Client devices provide a platform for the applications to run. Since applications are made of components, the client devices responsible for retrieving and executing the constituent components. They discover nearby peers and proxy server from which to request for components. They also incorporate caching mechanisms for performance improvement. In case a client does not have sufficient resources to run a component, as a last resort, it can decide to delegate the execution to a peer or an execution surrogate.

- *Facet servers*

  Facet servers host facets, i.e. components. They provide a permanent storage area for facets on the network. You can compare them to web servers which host web pages and cgi-programs.

- *Intelligent Proxies*

  The proxies are the main intelligence of the network. They receive requests from clients, match the requirements with description of the components and return the most appropriate components to the clients. The proxies also incorporate techniques to improve performance such as pre-fetching and co-operative caching.

- *Execution Surrogate*

  Execution surrogates are dedicated servers which run users' components. If a device does not have sufficient resources to run the next component, it can delegate the execution of that component to an execution surrogate.

4.3.2 Interaction Scenario

Every component fulfills a certain functionality. It is associated with a manifest which contains a description of its resource requirements, run-time behavior, context conditions, etc. An application, while running, needs components. It will request for a component which fulfills the required functionality, i.e. functional requirement. The resource manager in the client system adds to the request, the non-functional requirements such as the memory availability, processing power, user preference, and local time, if appropriate.

Under normal cases, the request is sent to the proxy. The proxy possesses the overall view – it knows what facets are available in the servers and it knows the requirements of the device. It matches the requirements of the client with the properties of the available facets, carries out some usage pattern analysis and determines which facet would be most suitable for the client to use. It then returns the facet to the client.

If connection to the proxy is not available, the client device can send requests to nearby peers to see if they have a facet with the required functionality available.

4.3.3 Functionality Adaptation

Dynamic component composition provides a flexible mechanism for achieving functionality adaptation. Functionality adaptation is made possible by three factors.

- The component model, which separates applications neatly into components, allows them to be composed at run-time, and to discard components which are no longer used.
- The resource managers in client devices, which maintain information about the physical resources, network connectivity and the context of the devices. This information is included in a component request and is the basis on which matching occurs.
- The proxies, which carry out matching between the request and facets. 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. This can be compared to Odyssey [23, 62] which achieves data adaptation by choosing among data with different fidelity levels. Hence, instead of choosing among different versions of data, we choose among different versions of functionality.

The main advantage of this approach, when compared to the approaches discussed in the previous section, is that it is very flexible and dynamic. 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, unlike 2K and DACIA. Which component comes in depends on the system and the matching mechanism of the proxy.

Also, in our model, the resource manager is contacted first to find out how much resource is available and then a component is chosen accordingly. In other models, a component is brought in, and then the allocation of the required resources to run that component is negotiated. In addition, since applications are linked by functionalities, rather than specific components, as new technologies or devices emerge, developers only need to write newer versions of the affected functionalities. The proxies will automatically match these components to suitable clients under the appropriate conditions. Rewriting or reinstallation of the whole program is not required.

Since the components are thrown away from the run-time after use, even the biggest of programs can be used in a small device, depending on the size and run-time behavior of each component. In short, this approach overcomes all the drawbacks in current functionality adaptation techniques, as mentioned in Chapter 2.

4.4 FACET MODEL

In this section, we describe the facet component model. 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 can be seen as providing functionality to carry out these tasks. These functionalities are embodied in facets. Facets are pure functional units. They provide a means to carry out the tasks, independent of the data or user interface (UI).

Many distributed systems use objects as a unifying abstraction for both data and functionality. Functionality is often bound with the specific data implementation it can act on. Grimm et al. [30,31,32] find such an approach flawed for pervasive and mobile environments. They argue that application functionality changes more frequently than data implementation and data layout. In addition, 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. Facets provide pure functionality. They take in inputs, 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 UI can be used to access the same functionality or tasks. In fact, the UI changes more often than the essential functionality of an application. When a new version of an application is released, the UI is often completely revamped whereas the basic functionality remains the same, except for a few additions and bug fixes. Since UI changes from device to device, 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.

In the following sub-sections, we provide an overview of the facet model – describing what constitutes a facet, how facets depend on each other to support execution and how they are requested from the proxy. We also discuss the container abstraction in which the facets execute.

4.4.1 Functionality

Before we can go into the details of the facet, we must clarify what a functionality is. As mentioned earlier, applications provide sets of functionalities to the user. These functionalities are independent of the user interface. *A functionality can be considered as a single well-defined task in an application.* The task could be as small as a matrix multiplication, or as big as blurring a whole image. It is mainly up to the programmer to decide what an application's constituent functionalities are or how "big" they are.

A more concrete definition of functionality is as follows: 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

- The set of input parameters, i.e. the number and types of the parameters.
- The set of output parameters, i.e. the number and types of the outputs.
- Description of what is carried out i.e. what are valid outputs for a set of inputs.
- Pre-conditions, if any, for example, the ranges of input parameters supported
- Post-conditions, if any, for examples, which values are nullified, error conditions.
- Side effects, if any, for examples I/O, or changes to state in the container.

Basically, 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 stick to the contract, they can be considered as 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 the contract which incorporates the factors mentioned above.

4.4.2 Facets

Facets are entities which implement the functionalities. They contain code components which follow the contract of their corresponding functionality. In our model, a facet implements only one single functionality. In other words, a facet cannot provide two or more functionalities.

Due to this limitation and the nature of functionalities, facets have the following two features

- *They have a single publicly callable method.*
  Every facet implements only one functionality. A functionality carries out a single task, which is accessed through a single interface with a defined set of input parameters and output parameters. Basically, a facet only has a single access point.

- *They have no residual state*
  What this means is that the functionality provided by a facet is independent of any previous invocations. During execution, a facet has a state which is determined by the values of the variables at that particular instant of time. Once the execution of the facet is finished, these variables are either discarded or are reset, so that they do not affect execution of the next invocation of the facet. A facet cannot keep any state beyond a single invocation. Thus, every invocation sees the facet as a "fresh" facet, without residues of previous invocation.

The above is of major consequence to the whole software model. These features of facets make them *throwable* – a facet can be discarded from the run-time as soon as it is used. Let's say a facet *does* have some residual state, for example, it contains some static variables which need to be maintained, and the result of the execution depends on these variables. In other words, every invocation to a facet depends on the previous state and will change it in some way when it finishes. In that case, that state between the two invocations must be maintained. And since this state is stored internally in the facet, it cannot be discarded. However, if a facet has no residual state, there is no need to maintain the state of the facet at the end of its execution. It can be discarded as soon as its invocation is over. If it is needed again, the same facet, or even perhaps another facet which implements the same functionality, can be reloaded. This results in a more flexible and memory efficient software model.

Every application does require a certain amount of state, i.e. data, or information, be maintained throughout the execution. Facets cannot be used to keep state which extend beyond one invocation. Therefore, application state is maintained in the container.

Consider the example of a facet which retrieves emails from an email server. This facet would require the user name, the password and the server from which to download the mails from. And after it retrieves the emails, it either stores them in the container or passes them on to the invoker. An "improper" or "stateful" facet, on the other hand, can be one which stores the user name, password and server name, internally on the first invocation and then for the

other invocations, does not require any inputs but rather uses the stored values. Or a "stateful" facet stores the emails internally after execution and just passes a pointer to them to the invoker (i.e. the email data is kept inside the facet). In this case, the facet cannot be discared after execution. If it is, then all the email data is lost. This goes against the notion of having throwable facets.

A facet can be uniquely identified by its facet id. Since a facet implements a certain functionality, it is associated with a functionality id (funcID), implying that it satisfies all the conditions of the contract specified for that functionality.

Externally, a facet is not completely a black box. It consists of two parts:

- *Shadow.* This describes the properties of the facet. It includes information about the facet for example the facet id, vendor and versioning information, the funcID of the functionality it achieves, its resource requirements (such as the size of the working memory) and its dependencies. More on dependencies will be addressed later. Basically, the shadow provides the meta-information about the facet. It is represented in human and machine readable form, XML, and thus can be accessed by developers, users and machines alike.

- *Code Segment.* This is body of the executable code which achieves the functionality. We utilize the object-oriented approach in our model. The code segment can consist of several classes. However, there is only one class which contains the publicly callable method corresponding to the functionality contract. The code segment is essentially a black box which exposes only one interface to access its service.

4.4.3 Facet Requests

There may be several facets implementing the same functionality. These facets are called *compatible facets*. Each of them could have different run-time characteristics, resource usages, etc. In other words, even though the facets may have the same funcID, they may have different properties. These properties are included in the shadow, and are taken into consideration when deciding which facet to use.

Whenever a facet is needed, the *facet specification* is filled in a facet request and sent to the proxy. The facet specification consists of the funcID of the functionality required and other criteria which the application developer may want to include such as the vendor, or version

information. The client system will add relevant context information such as the resources available, to the request before sending it to the proxy. The proxy essentially compares the criteria specified in request with the shadows of all the facets available and finds a facet which satisfies all the criteria and is suitable to run under the specified resource constraints. It then sends the facet to the client. The matching mechanism of the proxy is out of the scope of this thesis. Please refer to the Master of Philosophy Thesis by Vivien Kwan [52] for more information regarding the proxy.

4.4.4 Facet Dependencies

In order to achieve its functionality, a facet may call upon other facets. While executing, a facet could request for another facet. As mentioned earlier, the facet requests are in terms of functionalities, i.e. funcID, rather than for specific facets. In other words, a facet may depend on other functionalities. For example, an implementation of the gaussian blur functionality may depend on a matrix multiplication functionality.

Please note that functionalities themselves have no dependencies, it is the facets which have dependencies. Functionalities are independent of implementation. They are like empty boxes and hence do not depend on anything else. It is the facets, which fill in these boxes i.e. implement them, that may require the services of other functionalities.

Different implementation of a functionality may have different dependencies. Not every compatible facet has the same dependencies. One may require 2 other functionalities. Another may not have any dependencies at all and achieves the whole functionality internally.

Since the facets are actually composed and linked at run-time, every time a certain functionality is required, it is possible to get a different facet. Which facet is used depends on the run-time resource and context characteristics. And that facet, in turn, may have different dependencies. In essence, which facets are executed only can be determined at run-time.

To illustrate the above, let's say we have a facet *x* which depends on 2 functionalities: A and B. Each of facets *i, j, k* fulfill functionality A and facets *p, q, r* fulfill functionality B. A and B are called the dependencies of *x*. *Facet dependencies* are the *functionalities* a particular facet depends on.

At run-time, when we are executing facet *x* and it requests for functionality A, which of the facets *i, j, k* is brought in depends on the run-time characteristics and matching mechanism of the proxy. At one instance, it could be *i*. At another time it could be *k*. Even though *i, j, k* are compatible, they may have different dependencies. Again, which facets are called can only be known during execution. It cannot be predetermined. In addition, facet *x* may require A several times. The pictorial representation of which facets are actually executed at run-time, is called the *facet execution tree.*

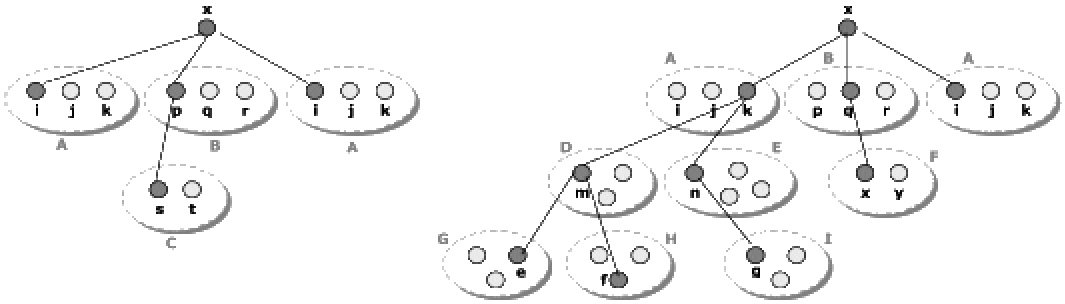


Figure 4.2 Different Execution Trees of Facet x

When a user invokes a certain functionality, it spans a whole tree of facet execution. The facet at the root of the tree is called the *root facet.* It is the facet in the whole execution tree, which is closest to the user interaction.

The facets, which are under execution at a particular instant in time, are called *active facets.* A facet is brought in and loaded when it is required. When it is under execution, it is active. As soon as its finishes execution, it becomes *inactive.* Inactive facets can be discarded to make room to bring in other facets. For instance, considering the execution tree in Figure 4.2, Facet *x* calls *k* which in turn calls *m* which calls *e* and *f*. After *m* is over, it calls *n*. At that instant, facets *x*, *k* and *n* are active, and facets *m, e* and *f* are inactive. They are not needed and thus, can be removed from memory.

4.4.5 Containers

Facets themselves do not keep any state and do not interact with the user. They only provide a certain functionality. However, most applications require maintenance of some sort of state and a user interface (UI). Facets are used by developers to build applications. They have a programming interface with which to communicate with each other. However, they cannot directly interact with the user. This is where the container comes in.

The container is an abstraction that provides an application-like feeling to the user. Users invoke the container to bring up an interface for user interaction, which in turn will request for the appropriate functionalities based on the user's input.

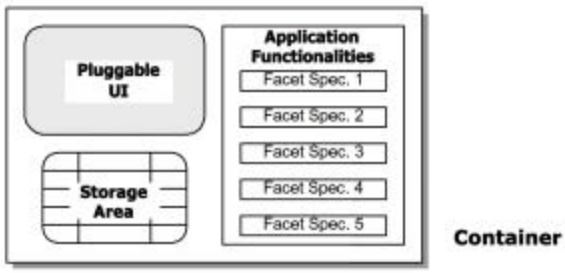


Figure 4.3 Structure of the Container

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. Facet specifications contain the funcIDs of the constituent functionalities and some additional criteria. When a particular functionality is required, the corresponding facet specification is sent as a request to the proxy, bringing in an appropriate facet and its dependencies, in turn starting off the execution of a whole tree of facets. Essentially these facet specifications can be considered as root facet specifications of the functionalities the application provides.

The container also incorporates a pluggable user interface. For different devices, there can be different UI code. The UI provides a means for users to access the functionalities offered by the container. It contains links to the set of functionalities in the container. When the user carries out a certain action, the UI will invoke the corresponding functionality by passing the facet specification to the underlying system to retrieve it and its execution tree from the network. Different UI code may link to different subsets of the functionalities, depending on the developer.

A container is much more than a bridge between the user and the facets. The main purpose of the storage area is to store execution state and data which is important for the execution of an application. These could be in the form of variables or objects. Since facets cannot be used to maintain any application state, this state information is maintained in the storage area.

The container also plays an important role in mobility. It keeps track of the state information, for examples, the execution status of the facets and shared data, and some relevant

information for restoring the execution. In short, we can say that the container is responsible for storing information about the execution state, in order to be able to restore it when the execution moves to another device. Exactly how that execution state is maintained and restored is out of the scope of this thesis. Please refer to the Masters of Philosophy thesis by Y. Chow [15] for more details regarding execution migration.

4.4.6 Comparison Among the Component Models

We have looked at some of the current component technology and the facet model. In this section we discuss how some of the issues affecting the different technologies and compare our facet model with them.

For the current component models, the components are either distributed together with the corresponding application to the client, or are located in remote servers. For the first case, again, we face the problem of limits placed on functionality an application can provide due to size constraints. The later case however, can be considered as services on the network. They are mainly heavy-duty server side components for applications which follow the client-server model.

Only in the facet model, the components are distributed individually to the client at run-time, when they are required. This highlights the mobility of the components in the facet model. Facets move to the client at run-time, whereas in the other models, the components are stationary. The only movement is perhaps a proxy moving to the client device enabling it to talk to the remote component.

In most of the component models, the components are dynamically linked. The types of dynamic linking can be broadly classified into two types, (1) interface-based and (2) object-based. For interface-based linking, components will link to each other via their interfaces. They will look for a particular interface id rather than a particular component. Which component is actually implementing that interface, does not matter. For object-based linking, applications look for particular components. They specify the exact name of the component or its instance. Object-based linking again can be classified into two types: One, the object or instance to be used is decided at design-time, and second, the object to be used is dynamically determined. For the first type, the name or identity of the component is actually built into the program, such as in COM, JavaBeans, EJBs and .Net Assemblies. CORBA falls under the second category of object-based linking. Even though it connects to a component via a

	COM	CORBA	JavaBeans	EJB	.Net Assemblies	Facets
Motivation	Reusable software components	Distributed object interaction	Reusable components for visual development	Reusable server-side components	Building blocks of applications	Functionality adaptation
COMPONENT PROPERTIES						
Language	Independent	Independent	Java	Java	Independent	Java
Platform	Mainly Windows	Independent	Independent	Independent	Independent[1]	Independent
Distribution Format	Exe or DLL	Executable, Car, or Aar	Jar	Ejb-Jar	Exe or DLL	Jar
Component Metadata	None	None[2]	Manifest	Deployment Descriptor	Assembly Manifest	Shadow
Identification	Class ID	Object Reference	Class Name	Class Name	Assembly Name	Facet ID
Characteristic Features	Components need to be registered with the OS	Clients need to be compiled with component's stub.	Components follow certain syntactic specifications	Components follow certain syntactic specifications	No need of globally unique names for private components	Components have one publicly callable method
CONTRACT						
Composition	Object-based	Interface-based	Object-based	Object-based	Object-based	Interface-based
Interface Definition	MIDL	OMG IDL	N.A	N.A	N.A	None
Number of contracts per component	More than one[3]	One[4]	One	One	One	One
Access points per contract	More than one	More than one	More than one	More than one	More than one	One
Component Linking	Run-time	Run-time	Compile time	Run-time	Run-time	Run-time
Location of Components	Local & Remote	Local & Remote	Local	Remote	Local	Local & Remote
Lookup Service	Registry/ Active Directory	Naming and Trading Service	None	JNDI	Global Access Cache	Proxy
Lookup Criteria	Class ID	Nickname or Service Type	Class Name	Nickname	Assembly Name	Functionality ID + other criteria
REMOTE COMPONENTS						
Remote Component Interaction	Proxy sent to client at run-time	Stub compiled into client program	N.A.	Proxy sent to client at run-time	N.A.	Component sent to client
Remote wiring protocol	DCE-RPC	IIOP	N.A	RMI-IIOP or CORBA	N.A	SOAP

1. .NET Assemblies are targeted for Microsoft's Common Language Run-time (CLR). CLR is theoretically platform independent. However, at present it is only available for Windows Platforms.
2. A CORBA object has no meta data. However a CORBA component contains an XML component descriptor
3. One COM component can implement more than one interface.
4. A CORBA object can have only one interface. However, a CORBA component can implement more than one interface

Table 4.1 Comparison of Different Component Models

specific object reference, the object reference can be dynamically determined by querying the Naming and Trading Service. A point to note about COM is that, even though all the interaction among components takes place via interfaces, component lookup is actually based on the identifier of the component, i.e. class id. Facets fall under the category of interface-based linking.

In the facet model, looking up of a facet involves more than just matching an interface identifier or a name. No doubt, we have to match the functionality identifier, which implies the interface of the facet. We also have to match the other criteria such as the resource constraints. Matching these often requires some in-depth analysis. The facet model, thus, requires a more complicated look-up service than the other component models.

There are many other differences between the facet model and the other models. These have been highlighted in Table 4.1.

4.5 RELATED WORK

In this section, we look at some projects which have employed component-based mechanisms in mobile and pervasive environments.

The Aura Group at Carnegie Mellon University [26] is looking into issues which affecting component-based development in pervasive environments. They consider software systems as collections of components co-operating to achieve a user's tasks. They argue that to be successful in a pervasive environment, component models must exhibit (1) mobility – tasks follow the user as he moves from one device to another, (2) adaptability – tasks can take advantage of resources as they change, and (3) resource awareness – components publish their resource requirements and offer multi-fidelity of services. Our facet model fulfills to a great extent their criteria of the component model. At the time of writing, their component model was not yet published, hence a comparison could not be made.

Yau and Karim [89] take a different approach to adaptation. Instead of having different versions of components with different run-time resource behaviors, they incorporate adaptability in a single component. Components are capable of performing self-customization so that they can fit themselves into specific real-time requirements. The components provide a set of built-in services which can be used to change the real-time

resource requirements of the component. However, since adaptation is carried out during the component integration stage of software development and not at run-time, it is not suitable for environments which experience a lot of changes at run-time.

Another project which considers applications as made up of functionalities is the One.World project [30, 31, 32] at the University of Washington. They argue that application data and functionality need to be kept separate in order for a system to be suitable for a pervasive environment. Functionalities are represented by components, which are dynamically linked and unlinked. An application's main component initializes its components and performs initial linking. While the application is running, it can initiate additional components, relink and unlink them as needed. The main difference between the facet model and One.World's model is the flexibility in the facet model. As mentioned earlier, in the facet model, which component is actually used is determined at run-time, as for One.World, which component to use is programmed into the application, thereby limiting the range of adaptability that can be achieved.

2K [47, 48] supports reconfiguration of component systems at run-time. 2K is actually a distributed OS rather than a middleware technology. However, it does endorse our approach of bringing in components only when they are needed. When a component is needed, the component and its prerequisite components are brought in from the component repository, which may be located locally or on the network. After that, the resource manager is contacted to allocate the required resources for the components. Developers have to provide specialized component configurator objects, which handle dynamic reconfiguration. If any changes occur in run-time resources, the resource manager will call the component configurator which will adapt the application, for example, by replacing a component by a new one.

The DACIA project [57], at the University of Michigan, provides a framework for building adaptive distributed applications. Applications are made of components located on various network entities and the links between the components represent the direction of data flow within the application. DACIA considers components as processing and routing units, transforming one or more input data streams. There are monitors, specific to particular applications, which are responsible for monitoring application performance and making configuration decisions. Applications are adapted by dynamically adjusting the connection between components and/or the location of the different components, and hence leading to a change in the application graph. This framework seems more suited for distributed applications which require data flow from one entity to another, for example video-on-demand.

For both 2K and DACIA, the adaptation policy is application specific. Every application implements its own adaptation policy, by means of component configurators (2K) or monitors (DACIA). This puts a lot of burden on 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. Also, this enables programmers to focus on application logic rather than on adaptation details, and makes it easier to develop mobile applications.

4.6 SUMMARY

In this chapter, we have seen that the current software distribution approach is essentially flawed when it comes to deployment in mobile computing environments. The *monolithic approach*, which distributes applications as one indivisible chunk, makes an application too big to fit into a small device, thereby, limiting the functionality a device can provide. In addition, the adaptability and the extensibility of applications are somewhat limited.

We then introduced the dynamic component composition approach. Applications are made up of components rather than as monolithic blocks. When a component is needed at run-time, it is brought in from the network and external. Once it is used, it can be thrown away. The advantages of this approach in mobile computing environments are that it does not limit the functionality of a device by its configuration. It also provides an extensive support for functionality adaptation and provides an increased the scope for peer-to-peer co-operation. In addition, it supports user mobility and migration adaptation.

The foundation of the Sparkle project is dynamic component composition. It is made up of several network entities which, together, form a mobile system with innate support for functionality adaptation. The main role in adaptation is played by the proxy. It matches the functional and the non-functional requirements in a client request with the components that are available and returns a suitable component to the client. Basically, functionality adaptation, in this approach, is achieved by choosing the appropriate component among different ones which have the same functionality. This provides a flexible and dynamic way to achieve functionality adaptation, which reduces the burden on the application programmer and provides for easy extension of adaptation alternatives.

In short, dynamic component composition is a suitable mechanism to provide functionality adaptation in a mobile environment. The components in the Sparkle Mobile Computing Environment are called facets.

The fundamental philosophy of the facet model is the separation of functionality from data and user interface. Facets are pure functional units. Every facet implements only a single functionality. A functionality can be seen as a contract – a single well-defined task in an application. It embodies the functional requirements of the contract, including the inputs, outputs, pre-conditions, post-conditions, etc.

Facets have two features, they only have a single publicly callable method, and they have no residual state. This makes them throwable enhancing the overall memory efficiency. A facet is made up of two parts – a shadow, which describes the properties of the facet and a code segment, which implements the functionality provided by the facet.

When an application needs a facet, it does not request for a specific facet. Instead it requests for a particular functionality and provides other criteria, such as resource constraints, to the proxy. The proxy finds an appropriate facet which satisfies all the criteria and passes it on to the client.

A facet may call upon other facets to fulfill its functionality. Facet dependencies are the functionalities a particular facet depends on. Due to the nature of the facet requests sent to the proxy, which facet is actually executed can only be determined at runtime. Active facets are facets which are under execution at a particular instant in time. Inactive facets can be discarded.

Facets execute inside a container. A container is an application-like abstraction. It has structures to store state. It also contains a pluggable user interface. A container is also a unit of mobility, in situations where the execution needs to migrate to another device.

This chapter ended by comparing the facet model with other component technologies and looking at some research carried out on components in mobile environments.

Chapter 5

The Sparkle Client System

We have discussed, in previous chapters, dynamic component composition, its importance in mobile computing, as well as the facet model, which is based on dynamic component composition. As part of my work, I have implemented a client prototype which supports the facet model in order to show the feasibility of such a model. The prototype supports bringing in facets at run-time, loading and linking them up with the execution, and after they are used, removing them from the memory. The client-system is built in the context of the Sparkle architecture described in Chapter 4. We have also built applications following the facet model which run on the client system successfully.

This chapter describes the Sparkle client system, its various constituent entities, their roles and implementations. Most of the mechanisms have been implemented above the JVM, in order to ensure portability, for example, the use of user-level class loaders to drop facets

5.1 CLIENT SYSTEM OVERVIEW

All the user devices involved in the Sparkle architecture will be installed with client system. The client system forms a basic necessity that all user devices must posses. Without it, the devices cannot access the services provided by the Sparkle system. Thus, it is vital to the whole architecture.

Because it will be installed on devices which come in all shapes, sizes and configurations, there are several requirements that it has to fulfill. These include:

- *Portability*

  There is no common configuration for devices on which the client system should work. The devices not only have different shapes and sizes, they have different

processors, different amounts of memories and even different operating systems. The client system should be able to work on the heterogeneous mix of devices.

- *Memory efficiency*

  Many of the devices may have small memories, often limited by their form-factors and power availability. Since the client system is a requisite for all devices, it should be kept as small as possible. It should also be able to dynamically change its run-time memory usage, if possible, in order to adapt to run-time needs.

- *Support for dynamic component composition*

  The client system should support all the intricacies of dynamic component composition. Most of the details are transparent to the programmer, hence the client system handles all the mechanisms involved. These include negotiating with proxies and peers for facets, loading and linking them up with the run-time when they are received, and discarding them when they are no longer active, etc.

- *Support for dynamic discovery*

  Devices move from one place to another. Their environments and nearby entities change according to their locations. Since dynamic component composition relies heavily on retrieving functionality from the network, the client system should have the capability of dynamically discovering nearby proxies and peers, in order to determine which would be best suited for facet retrieval at a particular instant.

- *Peer-to-peer co-operation*

  Devices need to interact with each other in several ways. They share information or files with one another. They can also share facets with peers in the vicinity, in a sense forming a pool of facets located close-by. A device can also delegate the execution of a facet to a near-by peer if it deems itself not having sufficient resources to execute that facet. The client system must be able to support all the different sorts of interaction possible between peers.

The current implementation of the client system addresses the first three requirements. We believe that support for dynamic discovery and peer-to-peer co-operation are inter-related and are complex issues. They require a separate in-depth study to look into the various possible mechanisms. The client is built on the Java Virtual Machine (JVM) in order to maintain the portability of the client system and the facets. A virtual machine is installed on top of the

device operating system, over which the Sparkle client system is implemented. The virtual machine forms the main execution engine of the system.

The client system accepts facet specifications, in the form of `FacetRequest` objects. It will then contact a proxy or a peer through the available network service 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, the system is responsible for throwing it away. 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.

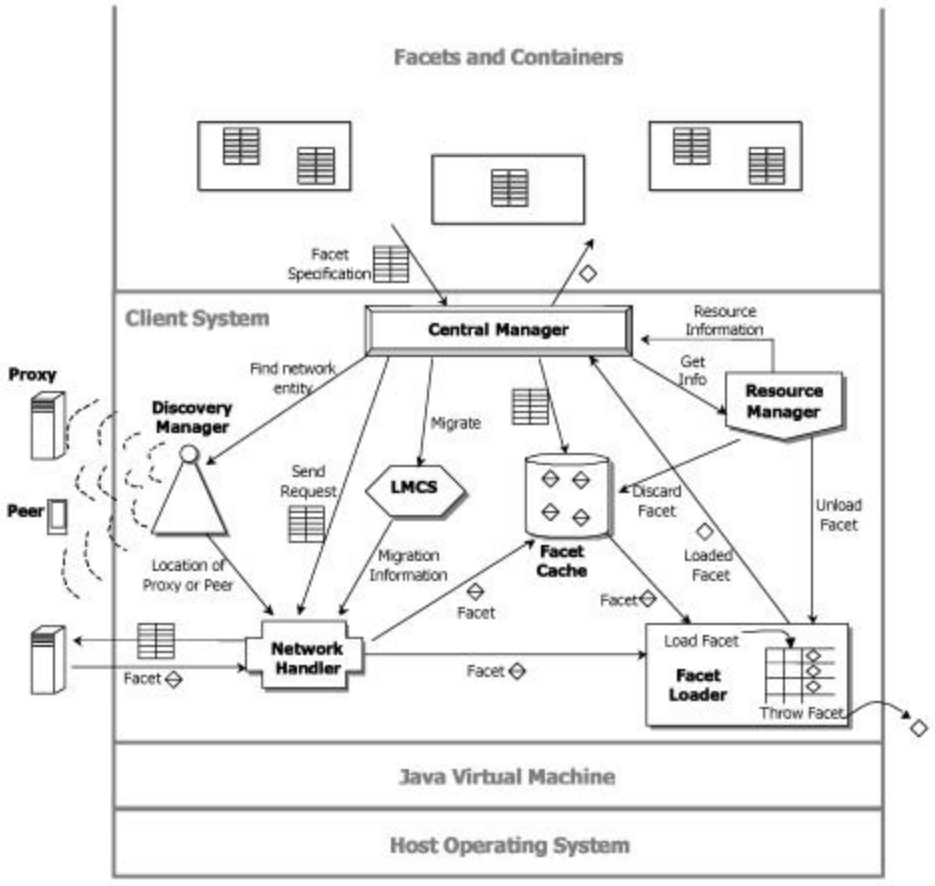


Figure 5.1 Architectural Overview of the Client System

Anatomy of the Client System

The Sparkle client system internally is made up of several entities, each of which have dedicated responsibilities. There is a central manager which co-ordinates the activities of all these entities in order to provide a unified system image. Facets and containers can only see and interact with the central manager. They cannot directly interact with the system modules. In this section, we describe the basic constituents of the client system and their roles.

The client system is made up of seven main entities:

- *Central Manager*

  The central manager is the main coordinating entity in the client system. It overlooks the activities of all the constituent entities in the client system. It interfaces with the facets and containers, accepts requests for facets from the user-level, delegates the appropriate tasks to the modules and returns a loaded, ready-to-use instance of the facet. It carries out the main housekeeping of the client system.

- *Discovery Manager*

  Facets are retrieved from the network. As a device moves from one location to another, it needs to keep track of nearby proxies and peers from which it can request facets. The discovery manager plays a vital role in this respect. It implements protocols and mechanisms to discover devices and network entities in proximity to the client.

- *Network Handler*

  A device needs to communicate with various entities in the surrounding network, for examples with proxies and peers. Each of which may employ different protocols. It is the responsibility of the network handler to handle the exact details of the communication mechanisms and provide a generic access mechanism for the central manager to use. It handles all the particulars of the protocol, negotiation, connection, sending requests and receiving facets, etc.

- *Facet Loader*

  When the network handler receives a facet from the proxy or other peers, it will pass it on to the facet loader. The facet loader is responsible for loading the facet into the run-time, creating a ready-to-use, invokable instance of it. Every facet has its own object space where its constituent Java objects are loaded, so as to facilitate discarding a facet.

- *Facet Cache*

  Instead of bringing in facets from the network every time a functionality is required, some of the commonly used facets are stored in the Facet Cache. Since the facets are locally available, this improves the response time and the performance at the cost of memory usage. The size of the cache can be dynamically changed, on the instructions of the resource manager. In addition, facets in the cache can be shared with peers. If a peer requests a facet and it is available in the cache, the client can send that facet to the peer.

- *Light-weight Mobile Code System (LMCS)*

  In a mobile environment, support for user and device mobility is indispensable. This involves moving the execution from one device to another, or perhaps delegating part of the execution to another device. The light-weight mobile code system (LMCS) is responsible for handling all the mechanisms involved, including suspending the execution, capturing the execution state, moving it to another device, restoring the state and continuing the execution.

- *Resource Manager*

  The resource manager keeps track of the resource usages of all the entities in the Sparkle system as well as the applications in the user-level i.e. facets and containers. It is responsible for resource determination and resource allocation. It can also control the amount of resources used by the entities in Sparkle system, for example, the facet cache. It appends to facet requests how much resources are allocated for those functionalities. It is also responsible for informing the LMCS to delegate the execution to another device. The delegation can be proxy-triggered – the proxy responds with a directive to delegate if a facet cannot be found which satisfies the specified resource restrictions. The delegation can also be client-triggered – while running a facet, if most of the resources are used up, the resource manager can stop the execution of that facet, rollback and ask the LMCS to migrate that particular facet.

As you can see from above, all the seven entities co-operate with each other when handling a facet request. They keep the underlying details transparent from the programmer so that he can focus on functionality and application logic rather than on networking details.

5.2 CLIENT SYSTEM ENTITIES

The previous section described the responsibilities of each of the client system's entities. In this section, we look into how these responsibilities translate into implementation terms. We have built a prototype whose main purpose is to illustrate the feasibility of dynamic component composition, rather than building a comprehensive mobile client system. Hence, the implementation effort focused on doing just that – dynamic facet composition.

Some entities require and in-depth study in certain area which is out of the scope of this thesis. For those entities, we have made use of simplified implementations. In the next sub-sections, we look at each of the entities in turn, the demands from them, how they have been implemented in the current prototype.

5.2.1 Central Manager

The central manager is center of control and coordination of the Sparkle client system. It controls the activities of all the entities in the client system. Because the Sparkle client system is small, with only six other entities, such a centralized model of control is feasible. If anything goes wrong in any of the other entities, they will inform the central manager, which can take the appropriate steps for recovery.

At present, it is the only entity which can directly interact with the user-level, i.e. the facets and containers. It receives requests to lookup a facet, puts the other entities into action to retrieve and load the facet, and will return an instance of the loaded facet back to the user-level. In case of no major errors, the sequence of actions taken by the central manager can be summarized by the flow diagram in Figure 5.2.

Ideally, the central manager co-operates with the resource manager in order to control the resources used by the various entities in the Sparkle system. At present, the resource manager provides simple resource management mechanisms and thus the duties of the central manager in the current implementation is limited to the flow chart in Figure 5.2.

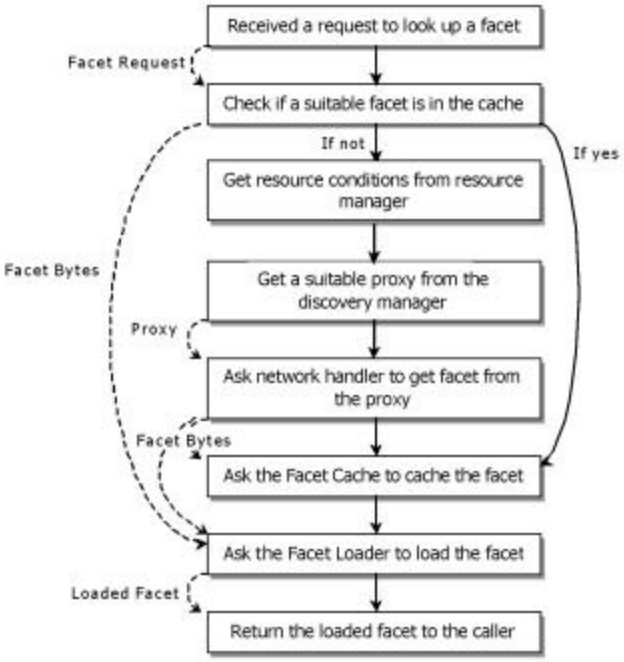


Figure 5.2 Actions Carried Out by the Central Manager

5.2.2 Discovery Manager

In a mobile environment, devices move from one location to another. A device may move from one wireless cell to another. It may move from one network to another, for example from a Bluetooth network to a GPRS network. Peers in the surrounding areas also change. The facet model depends a lot on retrieving facets from the network. Due to performance reasons, it is advisable for a device to connect to an entity which would have a faster download time. Consequently, the client must keep track of proxies and peers which are in close vicinity and determine which one should be used for facet retrieval. The discovery manager is responsible for discovering nearby entities. It implements discovery protocols. Different network environment may require different discovery algorithms. It is the responsibility of the discovery manager to employ a suitable protocols under the given network conditions.

The discovery manager plays an important role in the system. However, a comprehensive implementation of it would require an in-depth study of discovery protocols and a study of typical spatial patterns of devices and proxies. At present, our prototype does not support dynamic discovery yet. The location of the proxy is hard-coded in the implementation.

5.2.3 Network Handler

Clients need to communicate with various entities, including proxies and peers. Each of them may require different protocols for communication. For example, with proxies a client may use SOAP to communicate, where as with peers it may use a more lightweight, proprietary protocol. The network handler deals with all the details of communication with the different types of entities. It handles all the aspects including connecting and negotiating with the network entity, converting the facet request in a form suitable to be sent over the corresponding protocol. It also extracts the facet from the response and passes it on to the central manager. It handles encryption and decryption mechanisms, if required. In short, it provides a generic interface for other modules in the Sparkle system, such as the central manager, to interact with, keeping them shielded from the exact details of networking.

```
public class NetworkHandler {
    public byte[] getFacet(FacetRequest fr, NetworkEntity ne) {
    //requests for a facet from a particular network entity
    }
    .
}
```

Figure 5.3 Interface of the Network Handler

At present, the communication between the client and the proxy utilizes the Simple Object Access Protocol (SOAP). The facet specifications in the facet request are changed into XML format and sent over Hypertext Transfer Protocol (HTTP). An example of a sample request is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body> <ns1:GetFacet xmlns:ns1="FacetProxy">
     <facet>
       <functionality_id>20003</functionality_id>
       <vendor>SRG SANG</vendor>
     </facet>
     <rootfacet>no</rootfacet>
     <context>
       <user>
         <identifier>vjwmkwan</identifier>
       </user>
       <static_resource> ... </static_resource>
       <runtime_resource>... </runtime_resource>
     </context>
   </ns1:GetFacet></SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 5.4 Sample SOAP Request Sent to the Proxy

The request usually contains the criteria specified by the programmer as well as the resource availability information added by the resource manager. The proxy responds to the request by returning the matched facet as a MIME attachment to a SOAP response.

```
Content-Length: 2955
Content-Type: multipart/related; type="text/xml";
boundary="----=_Part_12_7542354.1019098911017"
SOAPAction: ""

------=_Part_12_7542354.1019098911017
Content-Type: text/xml

<soap-env:Envelope
xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
<soap-env:Body>
<ns1:GetFacetResponse xmlns:ns1="FacetProxy">
<facet href="cid:11289"/>
</ns1:GetFacetResponse></soap-env:Body>
</soap-env:Envelope>

------=_Part_12_7542354.1019098911017
Content-Type: application/java-archive
Content-Transfer-Encoding: base64
Content-Id: 11289

UEsDBBQACAAIAMFwiywAAAAAAAAAAAAAAAAJAAQATUVUQS1JTkYv/soAAAMAUEsHCAAAA
AAAAAFBLAwQUAAAACAB2eowsQn79FWsAAAB7AAAAFAAVAE1FVEEtSU5GL01BTklGRVNUL
.
.
UEsFBgAAAAAGAAYAowEAAEQFAAAAAA==
------=_Part_12_7542354.1019098911017--
```

Figure 5.5 Sample SOAP Response Received by the Client

5.2.4 Facet Loader

Once the network handler has extracted the facet from the response of the proxy, it is passed on to the facet loader. The facet loader then loads facet into the run-time and returns it to the central manager.

The current implementation makes extensive use of Java's user class loaders. Every facet is assigned its own class loader, a `FacetClassLoader` object, through which its classes are loaded into the virtual machine. The main motivation behind this is that it allows facets to be cleanly discarded. Normally, classes are not dynamically unloaded from the run-time. However, if there are no strong references to any of a facet's constituent classes or to its class loader, the Java Garbage Collector will consider them to be garbage and collect them. If a class loader is collected, then all the classes loaded by that loader will be unloaded from the virtual machine, consequently leading to the removal of the facet classes from the memory. In this way, a facet is "discarded".

The facet loader is given the facet and asked to load it. It will create a new `FacetClassLoader` object and delegate the task to it. The `FacetClassLoader` will load the main facet class – that is determined from the manifest of the jar file, and return an instance of it. The facet is kept in the `FacetClassLoader` so as to enable loading of the other constituent classes of the facet.

```
public class FacetLoader {
    public Class loadFacet(byte[] facetBytes, ClassLoader cl){
        FacetClassLoader fcl = new FacetClassLoader(facetBytes,cl);
        .
        Class facetClass = fcl.loadFacet();
        .
        return facetClass;
    }
    //other stuff
    .
    .
 }
```

Figure 5.6 The FacetLoader class

5.2.5 Facet Cache

The facet cache serves a dual purpose. Instead of having to bring in facets every time a functionality is needed, some of the commonly used facets can be cached locally. This would improve the performance of the system. The other purpose of the cache is to enhance peer-to-peer co-operation. The cached facets could be shared with nearby peers, which request for them.

Despite the advantages of the cache, it does, however, take up resources. The bigger the cache is, the larger its memory usage. Trade-offs need to be made. Since the resource manager keeps track of the resource needs of the whole system, it is responsible to determine what would be a suitable cache size for a particular time frame. The facet cache can change its size dynamically according to the instructions of the resource manager.

Currently, the facet cache is implemented as a two-level cache. The primary cache stores the facet in the memory. The secondary cache stores the facet as temp files in the secondary storage, i.e. hard disks for the case of laptops. Cache entries are replaced using the least-recently-used (LRU) scheme.

When the central manager receives a request for a facet, it will first forward that request to the cache to see if a suitable facet is locally available. Thus, the cache also requires a matching mechanism to identify which facet will be suitable. Since the suitability of the facets in the cache, in terms of resource usage, have already been matched by the proxies before, and due to processing limitations of the client devices, a simple matching mechanism is sufficient. At present, the cache tries to match either the functionality id or the facet identifier, with the shadows of the cached facets. If there is a match, i.e. a facet is found in the cache with the same functionality id or the same facet identifier, then the facet is retrieved from the cache

and sent to the central manager. In case of no match, the central manager will forward the request to the network handler to get the facet from the network instead.

```
public class FacetCache {
    public String cacheFacet(byte[] facetBytes) {
        //caches the facet bytes according to LRU scheme
    }

    public CacheEntry findFacet(FacetRequest fRequest){
        //finds a facet which matches the fRequest specifications
    }

    public void setLimits (int plimit, int slimit) {
        //changes the size of the primary and secondary cache
    }

    //other stuff
}
```

Figure 5.7 FacetCache Class

5.2.6 Lightweight Mobile Code System

In some sense, facets can be considered as mobile code segments. They move from servers to the clients when they are required (i.e. code-on demand). However, the responsibility of this module is not to move facets around, rather, to move the *execution* of facets. This is the module which enables user mobility. When users move from one device to another, they would want to carry on their tasks from where they left off. In addition, it is possible that the client device does not have sufficient resources to support the execution of a certain functionality. In that case, part of the execution can be delegated to another peer or a dedicated server.

The lightweight mobile code system (LMCS) is responsible for handling all the details of suspending and restoring the execution from one device to another. It needs to capture the state of the execution. Since one facet can call another facet, it needs to keep track of all the facets which are active under the current execution tree. It moves the container, the state information and the identifiers of the active facets to the other device. On the other device, the facets are retrieved from the proxy, and the execution continues where it left off. From then onwards, every time another functionality is required, it is adapted to the resource characteristics of the new device, rather than that of the original device. Such a mobile system is lightweight since it does not actually move the code from one place to another; rather, it moves the specification of the code, i.e. the facet identifiers and the state.

The actual implementation and the research issues involved are part of another study. For in-depth details about the LMCS, please refer to the Master of Philosophy thesis by Y. Chow [15].

5.2.7 Resource Manager

The resource manager, as the name implies, is responsible for the management of all the resources of the device. Resource management involves four aspects. First, determining how much resource is available at the particular instant in time. Second, determining the resource requirements of currently executing applications, or system entities. Third, evaluating and allocating resources, taking into account side effects and tradeoffs. Fourth, taking steps to enforce the allocation scheme. Resources encompass various aspects, such as memory, energy, processing power, network bandwidth, etc. In short, the resource manager tries to allocate the various available resources among the resource claimants, and enforces the allocation scheme, so as to attain a suitable working environment for the user and applications. The resource manager may need to keep track of previous resource usage patterns and resource states so as to derive an appropriate allocation scheme. It needs to be simple and effective. If it is too complicated, a device may spend too much time on resource management rather than on doing actual work.

In the Sparkle system, the resource manager determines how much resources, such as memory, network bandwidth, etc., are under use and by whom and how much resources are available. It allocates them accordingly to the various system entities and applications i.e. containers, running. For the system entities, it may directly inform them to change their resource usage. For example, the resource manager may inform the facet cache to reduce its cache size when running low on resources. As for the applications, the resource allocation scheme is enforced via facet requests sent to the proxy. Every time an application needs a facet, the resource manager appends the amount of resource allocated for the *whole execution tree* of that facet to the request sent. The proxy is then responsible for returning a facet which can execute under the specified resource constraints.

For every facet request, the resource manager will add the static resource constraints and the dynamic resource constraints of the device. Static resource constraints are those resources of a device which do not change with run-time, for examples, the display size, the processing power, etc. The dynamic resource constraints are those resources which can change with

every request sent to the proxy. These include, run-time memory size, network bandwidth and power, etc.

There may be situations in which it is impossible for a proxy to locate a facet which can run within the allocated resources. In that case, the resource manager can increase the resource allocation or, alternatively, it can decide to delegate the execution of that facet to another device or server. It will inform the lightweight mobile code system (LMCS), which will take care of all the details of the migration.

The current implementation of the resource manager supports rudimentary memory management. Our system is based on Java. The Java virtual machine only supports determination of the run-time memory availability. Incorporation of the management of processing power, energy and network bandwidth has to be built into the virtual machine, which may be the next step of the project. The resource manager, at present, implements a very simple allocation strategy. It allocates a constant fraction of the available memory to the facet to be requested. That allocated amount, together with the static resource constraints are added to facet requests sent out to the proxy.

5.3 DISCARDING A FACET

We rely on the Java Garbage Collector to discard the facet. The overall strategy is as follows – when a facet is no longer active, all the strong references to it are removed. Once the facet object is collected, there are no strong references to the facet classes and its class loader. The classes and the loader are collected as well, in effect discarding the facet from the memory. As mentioned in the previous chapter, when a programmer programs a facet, he essentially writes a `FacetImplementation` class. When a client requests a functionality, the `FacetImplementation` is received from the proxy and loaded. The underlying system will encapsulate the `FacetImplementation` in a `Facet` object. All calls to the `FacetImplementation` go through the Facet object.

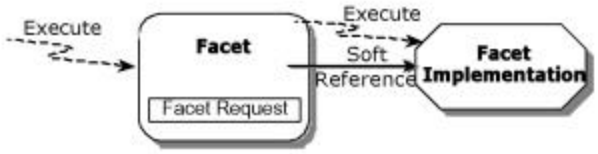


Figure 5.8 All Calls to FacetImplementation go through a Facet Object

The `Facet` object stores the `FacetRequest` of a functionality and a reference to a `FacetImplementation` object. When a client accesses a functionality, i.e. invokes the `execute method`, the `Facet` object will pass the `FacetRequest` to the manager, obtain the `FacetImplementation` object and call `execute` on it.

If there is no other reference to the facet, i.e. `FacetImplementation` object, it can be garbage collected. Since each facet is loaded in its own class loader, if a `FacetImplementation` object is collected, its class and class loader have no strong references to them. The garbage collector will get rid of the object, the class loader, the loaded facet classes. In this way, the facet is completely discarded from the system.

The `Facet` object and use four different types of references to deference the `FacetImplementation` object.

- *Strong References*

  After execute is called and the output is returned. The reference to the `FacetImplementation` object is still maintained. Hence, the `FacetImplementation` object cannot be collected by the garbage collector until the `Facet` object is collected.

- *Soft References*

  The characteristic of soft references is that they allow objects to be collected even though they have a reference to them. If an object is only accessible via a soft reference, we don't know when exactly it will be cleared. But we do know that it will definitely be cleared before the virtual machine runs out of memory.

- *Weak References*

  Weakly referenced objects are almost always collected by the garbage collector. If an object is only accessible via a weak reference, the garbage collector can collect it any time it wants.

- *Null References*

  Null references implies that no reference to the `FacetImplementation` object is maintained in the `Facet` object. Thus it can be collected as soon as it returns the output.

Often in an application, a facet is called upon more than once. If the `FacetImplementation` object has not been collected, then it is accessible via its reference

and can be immediately used. However, if it has been garbage collected, then the manager has to be contacted to retrieve the facet, either from the cache, or from proxies and peers. Thus, it is a tradeoff between memory and performance. If the facets are kept loaded in the virtual machine, then when they are needed again, there is no need to contact the manager again, improving performance, however the loaded facets would take up more of the virtual machine memory. The current implementation utilizes soft references, since it provides a reasonable compromise of both memory and performance. Please refer to Section 7.4 which describes some of the performance-related experiments carried out on the system.

```
public class Facet implements FacetInterface {
    ClassLoader cl;
    FacetRequest fRequest;
    SoftReference softFacet;
    FacetContainer container;

    public Facet(Object obj, FacetRequest fr, FacetContainer c){
      fRequest = fr;
      cl = obj.getClass().getClassLoader();
      container=c;
    }

    public Object[] execute(Object[] in){
      FacetImplementation fImpl = null;
      if(softFacet != null){
         fImpl = (FacetImplementation) softFacet.get();
       }

       if(fImpl == null) {
         //need to look up facet again from the manager
         fImpl = manager.lookupFacet(fRequest, cl);
         ..
         fImpl.setContainer(container);
         softFacet = new SoftReference(fImpl);
      }
      return fImpl.execute(in);
   }

   //other stuff
}
```

Figure 5.9 Facet Class

5.4 DISCUSSION

The main purpose of the prototype is to illustrate the feasibility of dynamic component composition and the facet model, i.e. being able to bring in a facet, link it up to the run-time and discard it after it has been used. Thus, most of the structures directly related to that have been comprehensively implemented, such as the facet loader. However, a complete client system requires much more than just that. It requires support for peer-to-peer co-operation, dynamic discovery and comprehensive resource management. More work needs to be done on the discovery manager, network handler, and the resource manager, in order to build a more complete client system.

Even though, at present, the resource manager is simple, it can illustrate the use of dynamic component composition to achieve functionality adaptation. The amount of resource allocated for a functionality is specified in a facet request. An appropriate facet is then brought in and executed. Thus, the execution of a functionality adapts to the resources available at run-time achieving functionality adaptation.

The main concern during the implementation of the client system was to keep all mechanisms above the JVM, i.e. keeping the internals of the JVM unchanged. This would keep the client system portable across various devices and JVM implementations. One of the main difficulties was to be able to unlink facets from the run-time system and discard them from memory. Ideally, the resource manager or the facet loader should be able to detect when a particular facet is not needed, and be able to decide when it should be discarded. However, this is not possible without making changes to the JVM referencing mechanism. Instead, we employed user-level class loaders. With these, when exactly a facet is discarded is under the control of the garbage collector. User-level entities, such as the resource manager, can just indicate whether a facet is available for discard by removing all strong references to it. To a certain extent, this method can achieve what we want, despite a slight loss of control.

5.5 SUMMARY

In this chapter, we have looked at the Sparkle client system. The client system is vital to the architecture since it is, in fact, the basic necessity that all devices must posses. The client system must fulfill the requirements of portability, memory efficiency, support for dynamic component composition and dynamic discovery.

The client side is made up of four layers - the operating system, on top of which is the Java virtual machine, then the Sparkle client system, and right on top is the application-level which is made up of facets and containers. The Sparkle client system is made up of seven main entities which are:

- *Central Manager*. The main coordinating entity in the client system, which interfaces with the applications.
- *Discovery Manager*. It locates nearby peers and servers.
- *Network Handler*. It handles all the details of communication between the various network entities.

- *Facet Loader.* It loads facets into the run-time and creates ready-to-use instances of them.
- *Facet Cache.* It stores facets locally to improve performance
- *Light-weight Mobile Code System.* It handles all the mechanisms involved in migrating execution from one device to another.
- *Resource Manager.* It determines the availability of resources and allocates them among the various run-time processes.

We looked at each of these entities in turn, and how they have been implemented in the current prototype. The current prototype illustrates the feasibility of dynamic component composition and the facet model. Discarding a facet was achieved by loading every facet with its own class loader, and either removing a reference to it when it is not being used, or using a soft or weak reference to it. When an instance of a facet is inaccessible by any strong reference i.e. there are no references to it, or it is only softly or weakly accessible, the garbage collector will collect the facet instance, its class loader and its corresponding class and, hence, discarding the facet.

Chapter 6

Programming for the Facet Model

In the previous chapter we looked at the implementation details of the client system. In this chapter, we look at the programmer's viewpoint. At present, the facet model is based on the Java programming language, mainly due to its inherent portability. Familiarity with Java or an object-oriented language will be beneficial to the understanding of the following discussion. Even though we are using an object-oriented language, facet programming is considerably different from traditional object-oriented programming. Facet programming has as its basis the separation of functionality and data, whereas object-oriented programming favors putting the two together.

In this chapter, we first describe the syntax for programming an application made of facets, including the facets, the container and the user interface. We then discuss the difference between facet-based and object-oriented programming, and provide guidelines on creating a facet-based program.

6.1 FACET-BASED PROGRAMMING

Creating an application based on the facet model involves understanding a couple of different abstractions. First of all, a programmer needs to know how to write the different parts of the facet i.e. the code segment and the shadow, and to pack them together. Since facets can depend on each other, programmers need to know how to request for another facet during execution. Finally, the programmer needs to know how to put all these together in the container so that the user sees a unified application, even though behind the scenes it is dynamically composed. We shall discuss all of the above in turn.

Applications and facets run on top of the Sparkle client system. The client system is designed in such a way that it takes care of all the dynamic composition of the facets. Networking,

negotiation and resource management details are transparent to the programmer, allowing the programmer to focus on the application functionality.

6.1.1 Facets

All facets implement only a single functionality. A direct implication of this is that facets can only contain a single publicly callable method. Facets are made up of two parts: a shadow which describes the properties of the facet and a code segment which implements the functionality.

6.1.1.1 Shadow

The shadow is actually an XML file, `shadow.xml`, with tags describing the properties of a particular facet. Some of the tags are compulsory, for example those indicating the facet ID and the functionality id. Others such as description or dependencies are optional. The proxy keeps track of the `shadow.xml files` of the available facets. When a proxy receives a request for a facet, it uses the shadows to find a suitable match for the request. Hence, a lot of tags in the shadow are used to provide meta-information to the proxy to carry out intelligent matching. The dependency information in the shadow is also used to prefetch the dependencies so as to improve performance.

The `shadow.xml` file can be divided into 3 parts. The first part provides information about the facet, such as its ID, vendor, version, the functionality it implements, an optional name and description, etc.

```
<identifier>GB00056</identifier>
<name>imageApp.imageFacet.GaussianBlur</name>
<vendor>SRG SANG</vendor>
<version>
  <major>1.0</major>
  <minor>a</minor>
</version>
<functionality_id>200007</functionality_id>
<description>Blurs an image pixel array </description>
```

Figure 6.1 Part of the Shadow Providing General Information of the Facet

The second part of the shadow describes its resource requirements. This could include anything from memory to network requirements. At present, the programmer must specify the memory requirements of a facet. It does not include the memory requirements of the

dependencies. He must include the static facet size, and the run-time working memory required. The run-time memory is, very often, a function of the input size. The programmer must specify the resource function with respect to the input size so as to assist the proxy in its matching decisions.

```
<resource>
<memory>
<static>128</static>
<dynamic>
   <input_variables>
      <parameter name="m"> 1 </parameter>
      <parameter name="n"> 2 </parameter>
   </input_variables>
   <formula> 3n^2+5m </formula>
</dynamic>
</memory>
<display>
   <width>300</width>
   <height>400</height>
</display>
</resource>
```

Figure 6.2 Part of the Shadow Providing Resource Information of the Facet

The third part of the shadow describes its dependencies. It specifies the functionalities the facet depends on, and whether the dependencies are necessary for the execution of the facet, or whether are used only in certain situations, i.e. optional. All this information is used by the proxy for resource matching and prefectching. The proxy recursively analyses the resource usage of the whole tree of dependencies, up to a certain level, before deciding which facet to return. In addition, since the dependencies of the facet is known, the dependencies can be prefetched from the facet servers in order to improve performance. When a request for the dependency arrives, it can be immediately dispatched to the client.

```
<dependencies>
  <dependency order="1" type="optional" subtype="if-then-else">
    <functionality_id>sparkle.imageapp.blur</functionality_id>
  </dependency>
  <dependency order="1" type="optional" subtype="if-then-else">
    <functionality_id> sparkle.imageapp.flip </functionality_id>
  </dependency>
  <dependency order="1" type="optional" subtype="if-then-else">
    <functionality_id> sparkle.imageapp.rotate </functionality_id>
  </dependency>
  <dependency order="2" type="compulsory">
    <functionality_id> sparkle.imageapp.matrix </functionality_id>
  </dependency>
</dependencies>
```

Figure 6.3 Part of the Shadow Providing Dependency Information of the Facet

6.1.1.2 Code Segment

The code segment is actually an implementation of a functionality. It follows the specification of the contract which includes the input parameters, output parameters, pre-conditions, post-conditions, etc. By definition, a facet only offers a single functionality,

hence it contains only a single publicly accessible method. That method follows the syntactic interface specified in the contract. A facet may have many constituent classes. However, there is only one class which exposes the publicly callable method. That class is called the *main facet class*.

Due to limitations put by the Java programming language and in order to simplify the implementation of the underlying Sparkle system, every facet is accessed through a uniform interface. This interface takes in an object array as input and returns an object array as output. If there is any error during the execution of the facet. It throws a `FacetExecutionException`.

```
public interface FacetInterface {
    public Object[] execute(Object[] in)
            throws FacetExecutionException;
}
```

Figure 6.4 The FacetInterface Class

When developing a facet, the main facet needs to extend the `FacetImplementation` class. The `FacetImplementation` class implements the `FacetInterface`, and provides methods for housekeeping which are used by the underlying system, and may or may not be accessible by the programmer. For example, every `FacetImplementation` object has a variable `fContainer` which indicates which container the facet instance is associated with. This variable is set when the `FacetImplementation` class is loaded by the underlying system.

```
public abstract class FacetImplementation implements FacetInterface {
    public abstract Object[] execute(Object[] in)
            throws FacetExecutionException;

    //other house keeping methods and variables
    protected FacetContainer fContainer=null;
}
```

Figure 6.5 The FacetImplementation Class

Here is an example of the main facet class which carries out Gaussian Blur.

```
public class GaussianBlur extends  FacetImplementation{
    //variables
    private static final int BYTE=0, SHORT=1, FLOAT=2, RGB=3;
    .

    //constructor
    public GaussianBlur(){
        .
    }

    public Object[] execute (Object in[]){
    //follows the contract
    //can call other methods in this class or
    //other classes in the facet
        .
        .
        return out;
    }

    //other methods
    void blur(PixelProcessor pp, double radius) {
        .
    }

    void blurFloat(PixelProcessor pp, float[] kernel) {
        .
    }
    .
    .
}
```

Figure 6.6 Example of a GaussianBlur Facet Implementation

Whenever a facet is invoked, it is invoked through the execute method. The numbers and the types of objects in the input and output object arrays are specified in the contract of the functionality. The execute method can invoke methods in the same class or in other classes which make up the facet.

The “stateless” nature of the facet must be kept in mind when developing a facet. A facet cannot keep any state beyond a single invocation. A facet only has access to the input parameters and objects stored in the storage area of the container it is running in. In addition, a programmer cannot assume that a facet will continue to “live” once the execution of the execute method has finished. It may very well be unloaded from memory, so any changes to static variables would be lost. Thus, every invocation must depend on the input arguments and the storage area solely.

6.1.1.3 Dynamic Resource Requirement

Once a programmer has written the code segment, he needs to fill in the details of the shadow. Most of the details are pretty straightforward. However, evaluating the dynamic run-time usage of a facet can be rather complicated.

The resource requirement information of a facet plays an important role in the whole architecture. It is the basis on which functionality adaptation takes place. The proxy compares the resource requirement of facets with the resource availability in the client. It will send a facet whose resource requirement and the resource requirements of all its dependencies put together is less than the resource availability.

Resource requirements can be considered in two dimensions, static and dynamic. What differentiates between the two is whether these requirements change at run-time. For example, the values or the amounts of static resource requirements do not change, whereas, the amounts of dynamic resource requirements may change depending on various run-time conditions, such as size of the inputs, algorithm etc.

Consider the example of memory usage. The code size (in kilobytes) of the facet is always constant. A device would need to have sufficient memory to store the facet when it is downloaded. Thus, this forms the static memory requirement. However at run-time, the actually memory that is used by the facet is highly dependent on the size of inputs and the implementation algorithm. For example, the memory used by a blur facet depends on the size of the image i.e. the size of pixel array which is the input to the facet, and depending on the algorithm, the memory requirement may be linear or exponential.

The facet programmer would need to specify both the static and the dynamic requirement in the shadow. The determination of the static requirement is rather straight forward, however the determination of the dynamic requirement requires some effort. We argue that most of the dynamic resource requirement can be expressed as a function of the sizes of the inputs. The requirements can depend on the sizes of more than one input parameter.

In order to find the dynamic memory requirements, the facet developer needs to test the facet with different sizes of input, and record the resource usages for each combination. He can run a mathematical or a statistical analysis program, such as Matlab, to go through the results and obtain a mathematical formula describing the resource requirements. That formula can be put in the shadow.

```
<memory>
  <static>233</static>
    <dynamic>
      <input_variables>
         <parameter name="m"> 1 </parameter>
         <parameter name="n"> 2 </parameter>
      </input_variables>
      <formula> 3n^2+5m </formula>
   </dynamic>
</memory>
```

Figure 6.7 Specifying the Memory Usage by a Formula

On the other hand, instead of specifying a formula, the programmer could include the resource usage as a look-up table for each combination in the shadow. However, the facet developer has to put an appropriate number of entries in the shadow. Including too many entries would make the size of the shadow and, hence, the facet, big. The number of entries should be sufficient to allow the proxy to do interpolation as required.

```
<memory>
   <static>233</static>
   <dynamic>
      <input_variables>
         <parameter name="m"> 1 </parameter>
         <parameter name="n"> 2 </parameter>
      </input_variables>
      <table>
         <entry>
            <input name="m"> 20 </input>
            <input name="n"> 10 </input>
            <value> 400 </value>
         </entry>
         <entry>
            <input name="m"> 40 </input>
            <input name="n"> 30 </input>
            <value> 2900 </value>
         </entry>
         .
         .
      </table>
   </dynamic>
</memory>
```

Figure 6.8 Specifying the Dynamic Memory Usage by a Lookup Table

Please note the resource requirements described in the shadow are the requirements of that facet only and not of its dependencies. Developers need to be careful to exclude the resources requirements of the dependencies when filling in the shadow. There may be cases in which the dynamic resource requirement cannot be expressed as a function of the input arguments, i.e. it depends on other factors, such as I/O, rather than on the size of the inputs. In that case, the developer can just consider the dynamic resource requirement as unknown.

6.1.1.4 Packaging the facet

A facet is made up of a `shadow.xml` file, and several class files which constitute the code segment. They are packaged together and distributed as a single entity. In our implementation, we use a JAR file to box them together. The facet JAR file is like an ordinary jar file with one main difference. In the manifest of JAR file, the programmer must indicate the main facet class. This will enable the Sparkle client system locate the `execute` method among all the classes in the JAR.

```
META-INF/
META-INF/MANIFEST.MF
shadow.xml
imageApp/imageFacet/GaussianBlur.class
```

Figure 6.9 Contents of the Jar file of the Gaussian Blur Facet

```
Manifest-Version: 1.0
Created-By: 1.3.1_01 (Sun Microsystems Inc.)
Facet-Class: imageApp.imageFacet.GaussianBlur
```

Figure 6.10 Contents of the Manifest of the Jar File

Our first choice for packaging the files was actually the Java Executable File Format (JEFF). JEFF is a new specification of Java file format proposed by JConsortium released in April 2001, which is specifically designed for small devices. By using JEFF, unnecessary duplication during class loading can be avoided. As claimed, JEFF can reduce the memory requirement to load a class by 40%. However, support for JEFF would require changes to the compiler and possibly the Java Virtual Machine. When we started our development, to the best of our knowledge, there was no fully implemented JVM with JEFF support. Thus, we utilize JAR as our packaging unit. At the time of writing of this thesis, a suitable JEFF implementation was not yet available.

6.1.2 Facet Requests and Invocation

Facets are brought in at run-time. That is the foundation of dynamic component composition. During execution, application code sends request for facets to the proxy, which will return suitable ones to the client. When a programmer requires a facet, he will fill in a `FacetRequest` object with the criteria he needs. These criteria from the *facet specification* of the required functionality. Most of them follow a key-value pattern. The programmer only needs to specify criteria to indicate the appropriate functionality. Details about the resource availability are added by the underlying system when the request is sent out to the proxy. The programmer does not need to worry about the resource constraints.

```
FacetRequest fr = new FacetRequest();
fr.addCriteria("functionality_id", "200007");
fr.addCriteria("vendor", "SRG SANG");
```

Figure 6.11 Adding criteria to a FacetRequest

The details of how a facet request is sent and how a facet is received is also kept transparent from the programmer. Once the programmer has filled in the `FacetRequest`, he just needs to declare a new `Facet` and invoke the `execute` method on it. The `Facet` class will interact with the underlying Sparkle system to load a facet implementation when it is needed.

```
FacetInterface fBlur =
    (FacetInterface) new Facet(this, fr, fcontainer);
.
Object[] input = null, output = null;
.
.
output = fBlur.execute(input);
.
.
output = fBlur.execute(input);
```

Figure 6.12 Invoking a Facet

As you can see from above, when declaring a new `Facet`, the programmer needs to provide certain information to the instance. It needs to specify itself (i.e. `this`), the caller. This is needed because the underlying system makes extensive use of user class-loaders. In order to ensure proper class loading and access of input parameters in the callee facet, it is necessary to determine the class-loader of the caller.

The programmer also needs to specify the `FacetRequest`. As mentioned earlier, the `Facet` class hides the details of when and how a facet implementation is brought in and also when it is discarded. It will transparently use the `FacetRequest` to retrieve the required facet when needed, if facet is not loaded (i.e. it is the first time the facet has been requested, or the facet has been previously discarded). In other words, a programmer just needs to declare a `Facet` object. He does not know whether that object actually contains a facet implementation at a particular instant in time. All he knows is that when he invokes the method `execute` on it, the `Facet` object will link up to a suitable facet implementation to carry out the functionality required.

The execution of every facet at run-time is associated with a container. Containers are application-like abstractions which provide areas for facets to run in and to store run-time data. When one facet invokes another facet, they both belong to the same container. Thus,

when a facet calls another facet, it needs to indicate which container it belongs to. More about containers will be discussed in Section 6.1.3.

A programmer accesses the services of a facet by invoking the `execute` method of the `Facet` object, with the appropriate objects in the input array. If there is any unrecoverable error during the execution of the facet, the method will throw a `FacetExecutionException`. In other words, facets interact with each other by method invocation.

6.1.3 Containers

Applications are implemented as containers in which facets can execute, store run-time data, etc. Since facets cannot interact with the users directly, containers form a bridge between the facets and the user by incorporating a pluggable user interface (UI). Every application has a set of functionalities it can offer to a user. When invoked, these funcitonalities form the root facets of the whole execution trees. Thus, the container stores the root functionalities it offers, the user interface, as well as providing a storage area to store shared data. In the current implementation, the set of facet specifications in the container is fixed. This implies that once a container is developed, the functionalities the application can provide cannot be changed. The UI follows the facet model as well in order to be consistent, i.e. it is invoked by the execute method. However, how it will evolve in the future has yet to be determined.

```
public abstract class FacetContainer extends Thread {
   public FacetContainer fContainer = this;
   public Facet UI;
   public RootFacets rootFacets = new RootFacets();
   public Storage storage = new Storage();

   protected abstract void setUI();
   protected abstract void setRootFacets();
   public abstract void run();

    public FacetContainer(){
        setUI();
        setRootFacets();
    }
}
```

Figure 6.13 The FacetContainer Class

When a developer want to write an application, he needs to extend the `FacetContainer` class and implement the `setUI`, `setRootFacets` and the `run` methods. When a container is loaded, the UI and the root facets are first set, and then the `run` method is called to start off the execution of the application.

RootFacets stores the functionalities this particular container can offer. The rootFacets object can be considered as an index table which holds the specification of the root facets. The programmer adds the FacetRequest objects into the rootFacets at a particular index in the table. This defines the functionalities a particular container provides.

```
protected void setRootFacets(){
        FacetRequest fr = new FacetRequest();
        fr.addCriteria("functionality_id", "200001");
        rootFacets.add(0,fr);

        fr = new FacetRequest();
        fr.addCriteria("functionality_id", "200002");
        rootFacets.add(1,fr);

        fr = new FacetRequest();
        fr.addCriteria("functionality_id", "200003");
        rootFacets.add(2,fr);
}
```

Figure 6.14 Defining the Functionality of a Container

If the UI code, or the container code need any of these root functionalities, they access them via their indices in the rootFacets object. When a functionality is retrieved from the rootFacets object, it is retrieved as a Facet object, ready to be "executed".

```
        Object[] input = new Object[]{pp, x, y};
        Object[] output =
                fContainer.rootFacets.get(2).execute(input);
```

Figure 6.15 Invoking a Root Facet

The main motivation of using an index-based access to root facets is that it maintains a certain level of constancy in an application. As different UI codes can be plugged into the same container, the UI developer does not need to deal with creating the actual request to retrieve the root facet. He just needs to link a certain user interaction with the functionality provided by a certain index in the rootFacets, provide the input arguments for the facet invocation and translate the output results into a user-comprehensible form.

Different UI implementers may link different subsets of the root facets. Say we have a container which can offer 8 functionalities (i.e. there are 8 entries in the rootFacets table). One UI implementation may link to all 8 of them, whereas another one only to 5 of them. Users of the second UI cannot access the other 3 functionalities. Such a scheme may seem awkward at first, but it provides flexibility to the programmer to handle user mobility. When a user moves from a resource rich device, to a resource constrained device with limited display ability, the whole container moves together with it. In the new environment, a more conservative UI can be plugged into the container instead of the previously used full-color UI.

It is then up to the UI programmer to decide whether to support all the functionalities provided in the container or to support only those he deems suitable.

UI code actually bridges the user interaction and the functionality offered by the container. At present, in order to keep things simple and consistent, the UI, in fact is just another facet, which extends the `FacetInterface` class and implements the `execute` method.

```
public class AppUI extends FacetImplementation {
    public Object[] execute(Object[] args){
        DisplayMenu();
        .
        .
        fContainer.rootFacets.get(2).execute(input);
        .
        .
        return null;
    }
}
```

Figure 6.16 Example of a UI linking to the Root Facet

The UI can be dynamically retrieved as well. The `setUI` method fills in a request. When the container loads, the request is transparently sent to the proxy and the appropriate UI is brought in. The container code can then execute the UI code. The current implementation uses the same mechanisms and API for the UI as it does for the facets.

```
    protected void setUI(){
        fr = new FacetRequest();
        fr.addCriteria("identifier", "ImageAppUI");
        UI = new Facet(this, fr, this);
    }

    public void run(){
       //container code
       .
        UI.execute(null);
       .
    }
```

Figure 6.17 Execution of the UI follows facet model

When a facet needs to access the storage area in the container, it does so via the `fContainer` variable which every loaded facet contains.

```
ImpageProcessor a = new ImageProcessor();
.
fcontainer.storage.add(4, a)
.
Object b = fContainer.storage.get(2);
```

Figure 6.18 Accessing the Storage Area of the Container

6.2 OBJECT-ORIENTED PROGRAMMING AND FACET-BASED PROGRAMMING

In the previous sections, we have briefly provided the syntactic overview of facet-based programming. Even though we have employed an object-oriented approach to implement facet, the fundamental principles of facet-based programming differ from object-oriented programming. In this section, we discuss some of the differences between the two and provide certain guidelines which programmers can follow when they build facet-based applications.

6.2.1 Object-Oriented vs. Facet-Based Programming

The fundamental difference between object-orient programming (OOP) and facet-based programming (FBP) is their centers of focus. OOP revolves around data. Programmers define not only the data structures of the data types, but also the operations that can be applied to a data structure. In essence, the data structure becomes an object, incorporating both data and the functions that can be applied to the data.

FBP, on the other hand, is focused on the application logic. Programmers define what functionalities an application provides, and these functionalities become facets. Facets do not store any data. Data structures are just input and output parameters to facets.

Objects encapsulate both state and functionality. At run-time, an object-oriented program's state is distributed among all the instantiated objects. The application logic is achieved by sending messages from one object to another, invoking different functions in different objects which in turn change the state in those objects. Facets on the other hand, encapsulate only the functionality. The application's run-time state is centralized. It is stored in the container, either in the storage or in the UI. Application logic is achieved by sending messages to the facets. The messages contain the data (state) on which the facets should act upon. Facets make changes to the state which is then stored back into the container.

The above is also reflected in the way programmers develop applications. The first step in OOP is data modeling – identifying all the objects they need to manipulate and how they relate to each other. The first step in FBP is functionality modeling – identifying all the functionalities in an application and how they are dependent on each other.

Since facets are types of components, the differences between components and objects also apply to facets and objects. A facet is a unit of third party composition; hence, it is sufficiently self-contained. Also, a facet has no persistent state. One copy of a facet cannot be differentiated from another copy. An object, on the other hand, is a unit of instantiation and has a state that can be persistent. It encapsulates both state and behavior. A facet may be realized by traditional procedures, assembly language or by using objects, as in our implementation. A facet may contain multiple classes, but a class is definitely confined to a single facet. The main difference lies in the roles of components and objects. The role of facets is to capture the static structure of an application whereas the role of objects is to capture the dynamic nature of systems built out of facets [82, 83].

Below we include a table, which highlights some of the differences between object-oriented and facet-based programming.

	Object-Oriented Programming	Facet-Based Programming
Unit of programming	Object	Facet
Granularity	1 class	Can have more than 1 class
Interfaces	Any number of interfaces, with any number of public methods	Only h as 1 publicly accessible method, which needs to follow a contract
State and Persistence	Stores some form of state during its lifetime. May contain some persistent state.	Does not store any state between 2 invocations. No persistent state in facets.
Driving Principle	Data-centric	Functionality-centric
Run-time Application State	Distributed among all instantiated objects	Centralized in container

Table 6.1 Difference between OOP and FBP

6.2.2 Developing a Facet-Based program

As mentioned earlier, developing a facet-based program is considerably different from developing an object-oriented program, and due to this reason converting an object-oriented program into facets is a tremendously taxing job. The main difficulty lies in the fact that in

OOP, data and application logic are intermingled together and are distributed all throughout the program, in different objects. In order to convert it to facets, data structures must be identified and separated from the application logic. The application logic must then be divided into facets. At the same time, it must be decided what data structures the facets act on. The UI aspects of the application must also be extracted and separately coded.

Instead of converting an object-oriented application into a facet-based application, it is recommendable to develop the application from scratch. Below we provide some guidelines for facet-based programming.

- *Functionality Modeling*

  The first step for FBP is functionality modeling – identifying all the functionalities in an application and how they are dependent on each other. The developer starts by identifying which functionalities or tasks the application should provide *to the user*. These functionalities essentially form the root facets of the application. Then, for each of the functionalities, it must be determined how they will be implemented, whether as a single facet, or whether the functionality should be divided up into smaller functionalities i.e. whether some of the work carried out to fulfill the functionality can be extracted and placed in another facet. The reason for doing this is that it allows reuse. For example, in an image processing application, both adding shadows to an image and finding the edges of an image require convolving a 3x3 kernel. Thus the convolving functionality can be extracted and implemented as a facet, so that both the "add shadows" and "find edges" facets can access it. In short, the developer has to determine the logic of the application and how it will be distributed among the various facets.

  There are no definite limits to the granularity of the functionalities or the depth of the dependencies. It is up to the developer to decide how fine they want their facets to be and how deep the dependencies run. As a general guideline, it is advisable to keep the dependencies shallow. It is because if the execution tree runs very deep, there are more active facets in the system. These cannot be discarded, thus clogging up the memory resources in a constrained device.

- *Facet Qualification & Programming*

  Then comes the nitty-gritty details of actually assembling the application. The data structures on which facets act, i.e. the input and output arguments, the pre-conditions, and post-conditions must be determined. Some facets may have to be programmed from scratch. However, some functionalities may already have been implemented before, by

other vendors or from previous versions. These can be used if they fit the requirements of the current development. The programmer may also provide different versions of a functionality. For example, different facets, carrying out the same functionality, implementing different algorithms and thus having different resource characteristics. These different versions are essential for functionality adaptation.

One thing to note is that if we are using an object-oriented language to develop facets, the inputs and outputs are in the form of objects. Thus, the data structures are encapsulated as objects and passed in and out a facet. These objects contain methods which act on their internal data. This may seem like a contradiction to FBP. However, looking at it more closely, these methods only provide functions to access the data, i.e. set, or get, or to do some minimal processing. The main application logic is located in the facets. These so-called data objects only have functions to make the access of the data structures more convenient.

- *Container & UI Programming*

  Facets can be considered as dispersed entities. The container is the unifying force, which brings the facets together so that they work together as a single application. The container stores the specifications of the root facets. It is the UI which provides the link between users and the root facets, and from one root facet to another.

  Since facets cannot directly interact with the user, it is the UI's responsibility to receive input from the users, convert into the appropriate data structures, call the corresponding root facets and then translate the output results and data structures in a form so that it is comprehensible by the user.

- *Testing & Distribution*

  Once all the facets and the container have been built, they must be tested together for correctness and to determine their resource requirements. The resource requirements, both static and dynamic, of each of the facets must be ascertained. The next step is to package the facets, with their shadows and distribute them to facet servers. The container and the UI can now be distributed to the users.

The above summarizes the steps which must be taken to develop a facet-based application. Throughout the development, it must be kept in mind that facet offer only a single functionality and cannot be used to maintain any state, thus the state must be maintained in the container.

6.3 SUMMARY

This chapter provided a programmer's perspective of the facet model. In the first section, the syntactic description of the facet model was discussed. The second section highlighted the difference between object-oriented programming and facet-based programming and provided guidelines for facet-based development.

Facets are made up of 2 parts, the shadow and the code segment. The shadow is just an XML file which describes the properties of the facet including information about the facet, the static and dynamic resource requirements, and the dependencies of the facet. The code segment is a group of classes implementing the functionality. Only one of the classes exposes a publicly callable method and that class is called the *facet main class*. Facets are packaged as JAR files and distributed.

Facet requests are carried out transparently to the programmer. The programmer just provides the facet specification. The underlying system will handle everything else, including bringing in the facet from the proxy, loading it and discarding it. The programmer is kept ignorant of all the details.

Containers provide a unified application abstraction to the user. They contain the specification of the root facets, and pluggable UI which provide links between the users and the root facets. One of the main purposes of containers is to provide programmers with storage area to store some run-time state.

The fundamental difference between object-oriented programming (OOP) and facet-based programming (FBP) is that OOP revolves around data and functions which act on the data. FBP is based on functionality and how it is organized among various facets. In OOP, state and application logic is intermingled together and distributed among the various instantiated objects. In FBP, application logic is separate from state and UI. Facets contain the application functionality, whereas state and UI are centralized in the container.

Developing a facet-based application encompasses the following steps:

- Functionality Modeling
- Facet Programming and Qualification
- Container and UI Programming
- Testing and Distribution

Converting an object-orient program to a facet-based program is a rather complicated task, due to the difference in the distribution of state and functionality throughout the program.

Chapter 7

Testing and Evaluation

The main implementation focus of my dissertation is the client system, which supports dynamic component composition. Chapter 5 described the client system and its constituent entities. This chapter provides an evaluation of the client system implementation in more concrete and practical terms.

The main purpose of the testing is to demonstrate the feasibility of the facet model and analyze its performance. Several test cases were created and run on the client system installed on a personal digital assistant. In this chapter, first, the testing methodology adopted is described. Then, we look at the features of the test bed. Testing was carried out under three contexts. We look at each of these contexts in turn, and describe the results obtained.

7.1 MOTIVATION

Our tests involve three aspects:

- To demonstrate the ability of the client to support dynamic facet composition, i.e. being able to bring in facets at run-time, load them, execute them and then throw them away.
- To establish factors, which can be adjusted to improve the performance of the client.
- To show that a real world application can be built by utilizing the facet model.

My dissertation covers only the client system, and hence, functionality adaptation, cannot be comprehensively tested. The proxy plays an important role in adaptation. The client system notifies the proxy of the amount of resources it processes, and the proxy then matches an appropriate facet for it. The evaluation of adaptation would require both the client system and the proxy, which is out of the scope of this thesis.

Performance is affected by many factors other than the client system implementation, including the network bandwidth and the implementation of the proxy. Thus, any performance data must be evaluated in the context of its environment, rather than attributing any shortcomings or strengths solely to the client system.

We carried out testing under three contexts. The facet model depends on the ability of the client system to request for facets, and load them at run-time. The first experiment looks into exactly that. It investigates the timing patterns of each of the stages involved in bringing in a facet and running it. We tested with facets of different sizes in order to identify bottlenecks.

The second experiment aims to identify factors which can have an effect on the performance of the system. We ran a benchmark application, which consisted of several facets and which called them in different access patterns, on the client and investigated the effect of caching, and the types of references on the performance of the application.

The last experiment aims to demonstrate the feasibility of building a normal application using the facet model. An image processing application was built, which provided several basic functionalities to the user. We wanted to investigate whether the delay in retrieving the facets from the network in the context of such an application can be considered acceptable by the user.

7.2 TESTBED

The Sparkle client system is built on Java. Thus, every client system requires a Java Virtual Machine. The size of the class files of the Sparkle system is 252KB, not including the dom4j XML parser which is actually 2.4MB. When installed together as a jar file which is compressed, the client system and the parser comes up to 630KB. The client system was installed on a Compaq iPAQ personal digital assistant, which was the basis of all the tests.

The iPAQ was directly connected to a Linux PC via a serial connection. A PPP connection was established over the serial line so that the Linux PC acted as a gateway to the Internet for the handheld. The maximum bit rate for the connection was 115200bps. The proxy was also set up on that PC. It stored facets and responded to client requests. This simplified proxy only matched facet identifiers and functionality identifiers in requests. It did not carry out any resource matching or adaptation.

Compaq iPAQ Pocket PC H3870		*PC Proxy*	
CPU	206 MHz Intel® StrongARM, 32-bit RISC Processor	*CPU*	Intel Pentium II MMX 300MHz
Memory	64 MB RAM + 32 MB Flash ROM	*Memory*	128MB RAM
OS	Familiar Linux v0.5.2	*OS*	RedHat Linux 7.1 (2.4.2-2 kernel)
JVM	Blackdown-1.3.1-RC1, native threads, nojit	*Web Server*	Apache 1.3.19-5
PPP Daemon	pppd version 2.4.0b4	*PPP Daemon*	ppd version 2.4.0

Table 7.1 Hardware Configuration used for Testing

The iPAQ is that it has 2 types of memory, 32MB of Flash ROM and 64MB of RAM. Data on the Flash ROM does not get erased after a reset and hence, it is used to store more permanent data such as the OS, the JVM, etc. Since the RAM is volatile, it is used to store temporary files and also is used as the dynamic working memory of applications. Currently, the OS and the JVM take up 16MB and 13MB of the Flash ROM respectively. After the installation of the client system, there is roughly 2MB of Flash ROM and 64MB of RAM left for applications.

7.3 EXPERIMENT 1 - TIMING ANALYSIS

The main purpose of the client system is to support dynamic component composition, i.e. being able to bring in facets at run-time, load them, execute them and then throw them away. In this experiment, we investigate the delay in bringing in and loading facets and look at the timing patterns, in order to identify bottlenecks.

Every application needs facets. These facets are located on the network. The client sends requests for these facets to the proxy, which returns appropriate ones to it. The main stages involved in the process on the client side include, (1) checking if the facet is in the cache, (2) creating a SOAP request and sending it to the proxy, (3) receiving the facet from the proxy, (4) caching the facet for future use, (5) loading the facet class into the virtual machine, and finally (6) creating an instance of the facet which can be used.

In this experiment, we ran an application which requested facets of different sizes and recorded the timing data for each of the stages. The code size of the facets ranged from 1KB to 500KB, however the actual size of the facets (i.e. the Jar files) ranged from 1KB to 69KB. This is because Jar files actually compress their contents when packing them together. The facets do nothing except output some strings to the console. For each facet, the application

was run 20 times and the average of the timing data was recorded. The cache size was set to 5 entries, and it was ensured that the cache was full when the request for the facet was made.

The timing breakdown obtained is depicted in Figure 7.1.

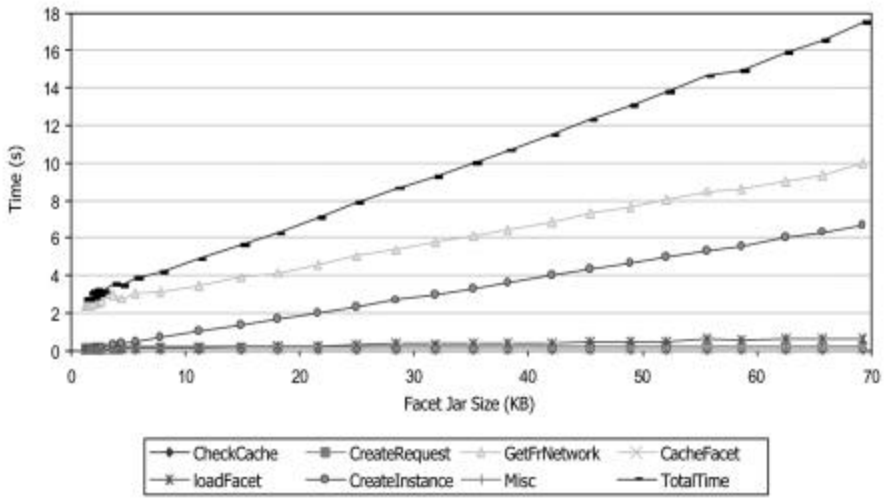


Figure 7.1 Timing Breakdown of Requests for Facets of Different Sizes

It can be seen from Figure 7.1 that as the facet size increases, the total time to retrieve a facet also increases. The stage, which takes the most amount of time, is the receiving the facet from the proxy. This is attributable to the slow network connection of the iPAQ. The latency of the network is approximately 2.3 seconds and the transmission rate is approximately 80kbps.

Another stage which takes up a considerable amount of the total time is the creation of an instance of a facet. For smaller code sizes, it is almost negligible. However, as the size of the facet code increases, it can be almost as much as one third of the total time taken. This value depends greatly on the implementation of the virtual machine and the processing power of the client system.

The loading of the facet is probably the stage which takes the third most amount of time. However, in comparison to receiving a facet from the network and creating an instance of it, this stage takes a relatively small amount of time. Again, this stage depends greatly on the implementation of the virtual machine and the processing power of the client system.

7.4 EXPERIMENT 2 – PERFORMANCE ANALYSIS

In the previous experiment, we noted that the main bottleneck was the time taken to retrieve a facet from the network and the time taken to create an instance of the facet. In this experiment, we investigate ways of reducing the cost incurred by that to an application, in order to improve performance. In this experiment, we look into two approaches, namely, references and caching.

An application requires facets. It may call upon a facet more than once. There may be three possibilities:

- *The facet is still loaded in the JVM.* It has not been collected by the garbage collector (GC) and is still accessible via its reference. Thus, it can be immediately used.

- *The facet is available in Sparkle's cache.* The facet instance has already been garbage collected. However, the facet code is available locally in the cache. In this case, the facet instance needs to be created again in the JVM before it can be used.

- *The facet is not available locally.* The facet instance has been garbage collected and the facet code is not available in the cache. In this case the facet has to be retrieved from the network, and the facet instance needs to be created. This incurs a great deal of overhead.

If the facets are never unloaded, even though this may improve performance, it takes up a lot of memory. Hence a balance between the three possibilities must be achieved. We investigated of using different types of references to dereference facet instances.

- *Strong references*

  If facets are strongly referenced, they cannot be discarded as soon as they are used, unless the references are explicitly nullified or the end of the scope is reached.

- *Soft references*

  If facets are soft referenced, then the garbage collector can collect the facet instance even though there is a reference to the instance. These instances will definitely be collected before the machine runs out of memory.

- *Weak references*

  These are similar to soft references. Only that the garbage collector will almost always collect the weakly referenced instances.

- *Null references*

  No references to the facets are kept. They become garbage as soon as they are used. This method can save a lot of the run-time memory taken up.

As it can be seen from above, soft and weak references are mid-way between strong and null references. For both soft references and weak references, the garbage collector can collect the referenced facet, as it deems necessary. If a facet has not been collected, it can be re-used, thus reducing the number of facet requests created.

We ran a benchmark application on the client systems several times, with different types of references, heap sizes, and cache sizes. The application consists of 21 facets. The code size ranged from 1KB to 32 KB with an average size of 10.7KB. However, the facet sizes (i.e. the jar file size) ranges from 0.9KB to 5KB with an average size of 2.6KB. The application called facets in different access patterns. The total number of facet calls is 724. A single facet may be called upon more than once, either in the same block or even by another facet. The execution trees of the root facets of the application had a maximum depth and breadth of 5.

In the first round of tests, we used different types of references and ran the benchmark on the iPAQ with the heap size set to 4MB and 16MB and the cache size to 8. The results are shown in Figure 7.2. For each test case, five iterations were carried out and their average value was recorded. Instead of measuring execution time, which is affected by a lot of factors, including bandwidth availability, processing power, garbage collection scheme, etc, we report how the requests for facets were satisfied. As mentioned earlier, when an application requires a facet, there are three possibilities for satisfying that request. Either the facet is located in the local cache, or it needs to be retrieved from the network. In the case that the facet has been used before in the same block, it is possible that the facet has not been discarded, i.e. it is still loaded in the JVM and the instance is still accessible. This is particularly the case for strong references. Comparing the results entails looking at the number of requests sent to the network, because that has a direct implication on performance. The more requests directed to the network, the longer the delay, and thus, the slower the performance.

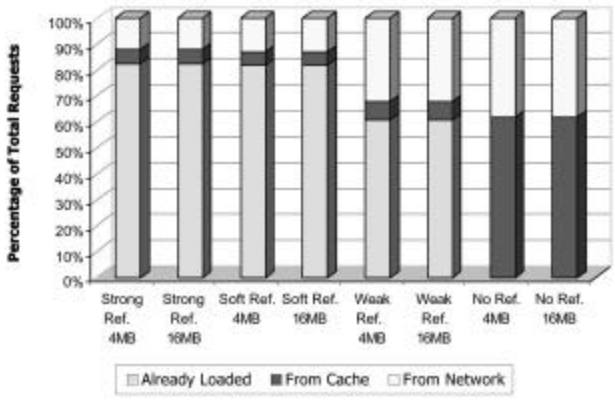


Figure 7.2 Results of the Benchmark with Different Types of References and Sizes of Heap on the iPAQ

As you can see from Figure 7.2, the results obtained somewhat match the expected results. The number of request which find the facets already loaded decreases from strong references to soft references to weak references to no references. Likewise, the number of requests sent out to the network increases from strong to soft references to references to no reference. There is a marked difference between soft references and weak references. However, there is little difference between the 4MB and the 16MB performance.

Theoretically, if less memory is available, the garbage collector will collect more soft references and the weak references in order to regain more of the heap. However, that did not seem to be the case in the above graph. This could be attributed to the particular implementation of the garbage collector. We carried out this experiment again, but on a PC which had the same configuration as the proxy and which was installed with Sun's Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_01). The results are depicted in Figure 7.3. In this case, a difference can be seen between in the performance of soft references and weak reference, in a smaller memory situation i.e. 4MB heap size, and in a high memory situation i.e. 16MB. With less memory available, soft and weak references are more easily collected, and hence it is less likely to find the facet still accessible through the reference. In other words, more requests need to be directed to the cache or to the network. The heap size has a bigger effect on the case of soft references than that of weak references.

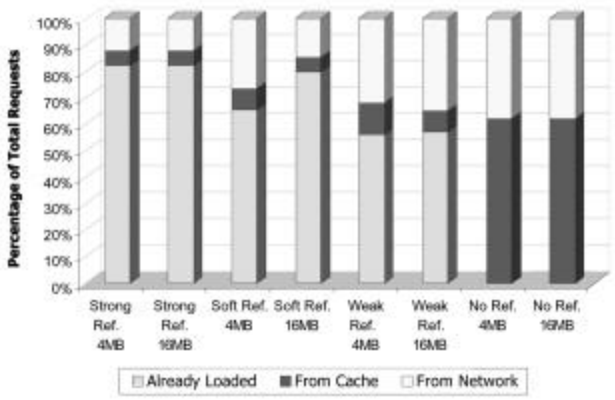


Figure 7.3 Results of the Benchmark with Different Types of References and Sizes of Heap on a PC

From the graphs above, it can be seen that soft references are probably best suited for the client system. Strong references disallow the discard of facets, which goes against the "throwable" philosophy. The benefit of weak references when compared to no references, in the presence of a cache, is not as significant. That is, there isn't much difference in the numbers of requests retrieved from the network. Soft references, hence, are a better option for our system. They provide performance improvement, and adaptation to different memory conditions, depending on the implementation of the Java Virtual Machine. Their performance is comparable to that of strong references. When running out of memory, the soft referenced facets will be discarded, reclaiming the much needed memory. It must be emphasized that the results obtained above are greatly affected by the implementation of the garbage collector. However, the tests above serve as indicators of what can be expected from different virtual machines, and can be used to make design decisions.

In the second round of tests, we investigate the effect of caching. The system caches the recently used facets, i.e. it implements the Least-Recently-Used (LRU) algorithm. It will first look into its local cache to see if a facet is available there, if not, only then will it request the facet from the network. In that way, it reduces the number of requests sent out to the network, and thus improves performance.

We ran the benchmark several times on the iPAQ with different cache sizes. Soft references were used and the heap size was set to 4MB. For each test case, five iterations were carried out and their average was recorded. The results are illustrated in Figure 7.4.

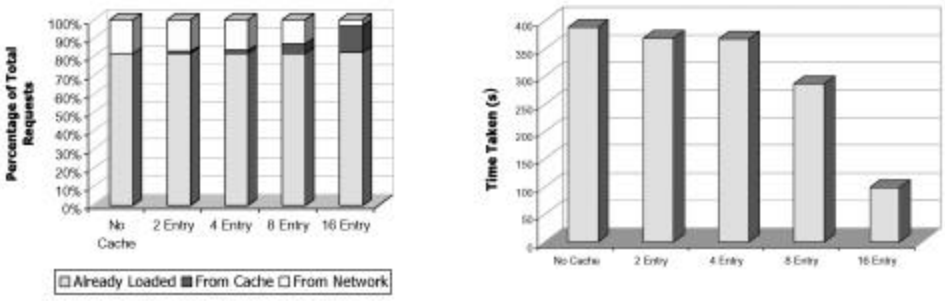


Figure 7.4 Effect of Caching on Performance

It can be seen from the graph, that as the cache size increases, less requests need to be sent out to the network and thus improving performance.

7.5 EXPERIMENT 3 – IMAGE PROCESSING APPLICATION

In this experiment, we illustrate the feasibility of employing the facet model to build a usable application. We have built an image processing application, which provides basic functionality to the user. The application consists of 15 facets, each providing different functionality, out of which 10 are root facets.

The structure of the application is depicted in Figure 7.5. The root facets essentially provide functionality to open, to blur, to find edges and to flip images. The other facets provide functionality such as matrix convolvers and converters (i.e. converting an image of which the pixels are in bytes into an image with pixels as shorts).

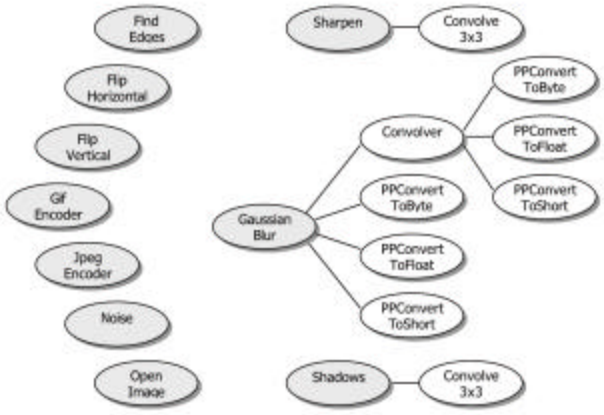


Figure 7.5 Facets of the Image Processing Application. (Gray ovals represent root facets)

The size of the application container and UI is 10.1KB. The total size of the facets is 51KB, the average being 3KB. We carried out two tests on the iPAQ handheld. One to measure how long it takes to retrieve a facet and how much of that time is spent on receiving it from the network. In the second test, we ran the application and compared the difference in response times, with respect to the user, of each of the functionalities, when the facets are locally retrieved, i.e. from the iPAQ itself, or from the proxy. The graphs of the tests are depicted in Figure 7.6. The values in the graphs represent averages of 20 iterations.

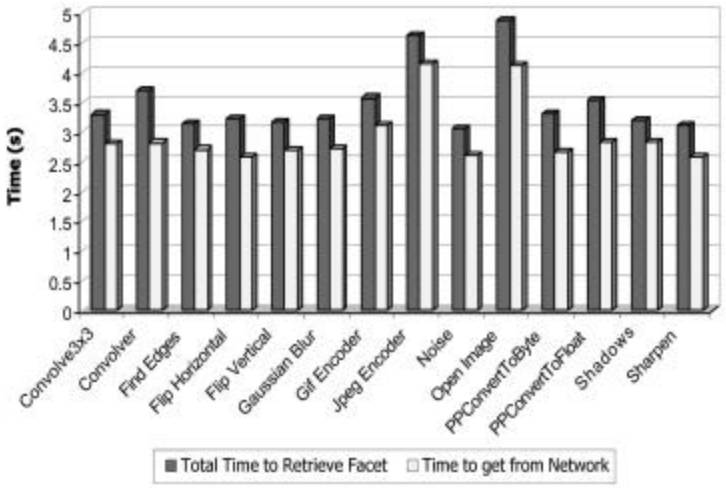


Figure 7.6 Timing Analysis of Retrieving Various Facets from the Network

The average time to retrieve a facet is 3.5 seconds, out of which most of the time is spent waiting for the network transmission, which takes, on average, 2.9 seconds. This corresponds with the results obtained in Experiment 1, in which it was indicated that the main bottleneck was the network transmission.

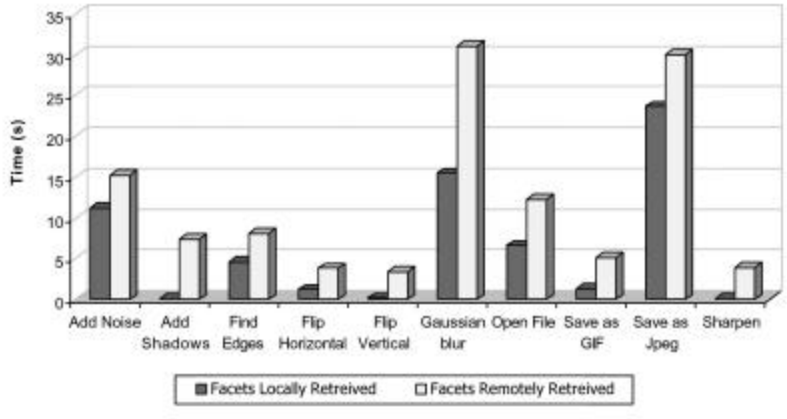
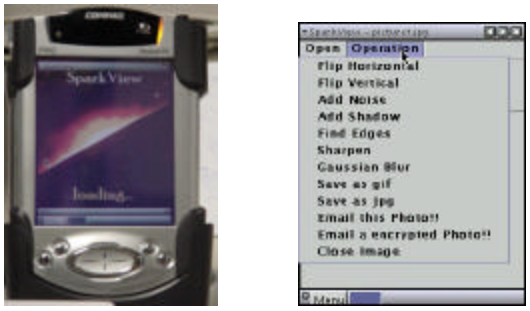


Figure 7.7 Comparison of Response Times for Facets locally retrieved and remotely acquired.

Figure 7.7, we compare the response times in executing a functionality when the facets are locally available and that of retrieving them from the network. The functionalities were invoked on the same image. The difference between the response times is dependent greatly on the number of dependencies the root facet has, and whether those dependencies are available in the cache or not. For example, a huge difference can be seen in the execution of Gaussian Blur. This is because the Gaussian blur facet has the most number of dependencies all of which must be retrieved from the proxy, leading to the accumulation of the transmission delays of the facets. The average difference in response time between having all facets locally available, and that of having to retrieve all of them from the network is 5.6 seconds. With no mechanisms incorporated for performance improvement, such as caching, this can be considered as acceptable.

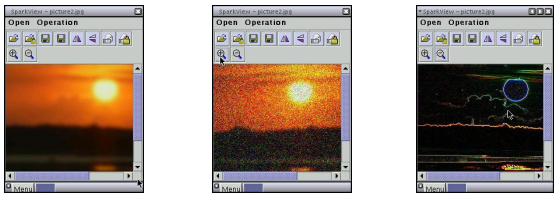
The screen shots of the application are depicted in Figure 7.8.



The application on the iPAQ **Functionalities provided**

Figure 7.8 Screen Shots of the Image Processing Application

Original Image



After Gaussian Blur **After Adding** **After Finding Edges**

Figure 7.9 Applying Various Functionalities on an Image

It can be seen that the facet model can be used to build practical applications, especially those which contain a lot of functionalities. Mechanisms, such as caching, or pre-fetching algorithms can be incorporated which reduce the apparent delay and improve the response times.

7.6 EVALUATION

From the above experiments, it can be seen that the main bottleneck is the delay caused by network transmission. We cannot reduce the latency per se, however we can improve the performance by reducing the number of requests sent to the network. One mechanism, which was explored in the second experiment, was the use of references. Instead of marking every facet as garbage as soon as it has been used, i.e. by removing references to it, we can let it be accessible by soft references. In that case, if the client has sufficient memory, the facet will not be collected, i.e. discarded, and will still be accessible. If it is required again in the same scope, it can directly be used, without having to load it or to create another instance of it.

Another mechanism which was explored was the caching of the facets. Facets were locally retrieved instead of getting them from the network. At present, the LRU algorithm is being used for caching. An increase in the size of the cache demonstrated an exponential decrease in the number of requests sent to the network and in the execution time as well.

The above are mechanisms which the client side can implement on its own. Other techniques, which require co-operation with the proxy, can be also employed. For every request, the proxy will carry out dependency and resource analysis. It can predict, to a certain extent, which facet will be used next and prefetches it so that it is ready to transmit when the next request arrives. Instead of waiting for the next request, a push mechanism can be utilized which allows the proxy to send the facet to the client *before* it is requested. In other words, while the client is executing a facet, the proxy pushes the dependencies into the cache, so that when the dependency is required, the client can retrieve it from its cache instead of requesting the facet from the proxy. This technique reduces the response time and can improve performance. At present, the above technique has not been implemented. This and other techniques are avenues for future work on the system.

The final experiment illustrated that the facet model can be used to build practical applications. It is believed that the facet model is best suited for applications which have a lot

of functionalities, which can benefit from the memory saving and dynamic adaptation provided by the facet model. The image processing application suffered some response time delay due to the network transmission of the facets. However, for facets which are already located in the cache, the performance delay is insignificant.

It must be kept in mind that the current experiments were carried out on a serial connection with an effective transmission rate of 80kbp. With wireless LANS and bluetooth networks which have a transmission rate of 54Mbps and 1 Mbps respectively, the network delay may be greatly reduced.

Even though our testing only focused on the client system, we envision that the performance of the whole sparkle system will not be much worse. The current test results do not include the time taken for the proxy to carry out analysis and matching. However, since the proxies are implemented on machines with sufficient processing powers, we believe that the delay incurred will not be as substantial as the network delay, and may be in the order of tens or hundreds of milliseconds.

7.7 SUMMARY

This chapter described the testing carried out on the client system. The main purpose of carrying out the experiments was to demonstrate the feasibility of the facet model. This was done in three contexts. The tests were carried out on a Compaq iPAQ personal digital assistant.

The first experiment investigated the timing patterns of each of the stages involved in bringing in a facet and running it. The second experiment identified factors which can have an effect on the performance of the system such as the use of references and caches. The last experiment demonstrated the feasibility of building a normal application using the facet model.

From the first experiment, it was concluded that the main bottleneck during the request for a facet was the network transmission delay. In addition, creating an instance of a facet took a significant portion of the time in cases of large code size.

The second experiment demonstrated that performance could be improved by using weak references, soft references or strong references, because they reduce the number of requests for facets and in turn the number of requests sent to the network. However, strong references disallow the discard of facets, and the benefit of weak references when compared to no references, in the presence of a cache is not as significant. Soft references, hence, are a better option for our system. They provide performance improvement, and adaptation to different memory conditions, depending on the implementation of the JVM.

The third experiment demonstrated the feasibility of building a practical application using the facet model. An image processing application was built which provided functions like opening, blurring, flipping and saving images, etc. It highlighted the need to incorporate mechanisms to improve response times.

At present, only two mechanisms, soft references and caching, have been incorporated into the system for performance improvement. Other techniques, which involve the co-operation of the proxy to reduce the observed latency in receiving a facet, can provide avenues of future work.

Chapter 8

Overall Discussion

The last few chapters have focused on the facet model, the details of its implementation and that of the Sparkle client system. The previous chapter described some of the tests carried out on the prototype of the client system.

In this chapter, we take a step back and look at the overall picture. Despite having gone through a lot of issues, there are some which have not been discussed and deserve a mention. We particularly look into web-services and the distinctive features of the facet model. Web-services have been hailed as the computing paradigm for the future, and thus it is important to know where the facet model stands in the face of web-services.

We also look at some issues regarding the facet model itself. No doubt, there are certain cases in which the facet model takes a different approach than the others. When it comes to functionality adaptation and resource management, the facet model opts for programmer transparency, whereas in many other systems, the programmer has a large part to play in resource management. This chapter will also look into the transparency issue.

The main motivation of the facet model is its ease of adaptability. We analyze how effectively the facet model can be applied to achieve the different types of adaptabilities discussed in Chapter 2. We also look into how context-awareness can be achieved under this model.

Nothing is perfect. Everything has some flaws and strengths. We discuss some of the deficiencies of the facet model and then describe situations in which it would be most suitable.

Finally, we look at the Sparkle architecture as a whole, and describe what it is lacking in order for it to be a suitable architecture for the mobile computing environment.

8.1 WEB-SERVICES VS FACET MODEL

The Internet and the World-Wide Web have traditionally been used for access of information, web pages, emails, etc. However, recently, there has been an expansion of the paradigm to include the provision of web services. The web services movement has a lot of major software vendors as its proponents including Microsoft (.NET), IBM (WebSphere), Sun Microsystems (SunOne), Oracle (Dynamic Services), and HP (HP Web Service Platform). It has also been the focus of a lot of research including Ninja [29], Sahara [69], Open Mash[1] etc. The movement is about using the Web as a platform to provide services.

Web services can be considered as *active programs or software components, which are hosted on the web-servers and can be accessed by other entities*. What sets them apart from cgi-programs is that they can be discovered and composed. A web service is usually associated with a description using WSDL, Web Services Description Language, and is published in a directory such as the Universal Description, Discovery and Integration Service (UDDI). This allows the discovery of the service. Various services located on different servers can be dynamically composed together to provide a complex service to the user. SOAP is the *de facto* protocol used for invoking services.

Considering the client side view, functionality is implemented by services which are hosted on third-party servers. Users send data, or information about the location of the data, to the services which then act on the data and send the results back to the user.

The web services approach partially alleviates the heterogeneity problem and the adaptation problem. Most of the essential parts of the application are executed on servers, whose configurations and runtime conditions are known. This makes adaptation of the application redundant, because the most resource consuming part of an application is carried out on the servers.

However, it is possible that the results returned by the services may not be in a form suitable for the client devices. For example, they may be encrypted with a large key and it would take a lot of computational effort for the device to decrypt them. Or, the results may be too large to send over a slow wireless network to a device. This necessitates the use of a transformational intermediary.

The transformational intermediary is often implemented as a proxy responsible for adaptation. It carries out data adaptation - transforming the results returned by the services. For examples, changing HTML results to WAP, or changing the image resolution. The proxy may also carry out network adaptation, such as using protocols especially designed for low power, low computation or poorly connected devices as the need arises. Thus, the proxy becomes the center for adaptation. In the facet model, the proxies also have a major role to play in adaptation. However, their role is focused mainly on functionality adaptation.

The fundamental difference between the facet model and web-services is that the facet model brings code to the client to execute, and in web-services, data is moved to the servers and the results are sent to the client. Both models have their advantages and disadvantages. We cannot say which is better than the other because both systems have to be more widely adopted in order to provide a more comprehensive comparison. Below, we highlight some issues which are of consequence to mobile computing.

As mentioned earlier, the facet model brings in components rather than accessing the application logic remotely, i.e. web-services. The latter method requires moving data to the service, and getting back the results. It would be easier to move components rather than data because, very often, the code size is often smaller than the data size, especially in the facet model, where each component only supports a single functionality. Also, there is less of a need for marshalling and unmarshalling the data, thus reducing processing time.

Moreover, there may be situations in which it is difficult to send data to the service, either because the data is too large to be sent over slow wire-less networks, or because the data is private and should not be sent over an insecure network to a third-party server. In those situations, there is no option but to bring in code.

In addition, devices need to be able to access the services from any place, at any time. This places a burden on the service provider who must ensure service availability. If a service fails, multitudes of devices may not be able to function, unless another compatible service is available. On the other hand, in the facet model, if a proxy fails, a client can always access another proxy. If a facet is not available, a client can get another compatible one instead.

The web-services approach strictly follows the client-server model. It requires that all interactions that occur are those between a client and a server. Any communication between two devices has to go through an intermediary service. Devices cannot directly communicate with each other. This model does not allow for direct client-device peer-to-peer

communication, which is one of the characteristics of mobile computing. For both performance and security reasons, it is not desirable for all peer-to-peer communications to be carried out via intermediary servers.

Furthermore, in some cases, it may not be possible to carry out the application logic on servers. Some things have to be done locally. One such example is the processing of private data which cannot be moved away from the device. Another example is a miniature robot on an exploration journey, inside a digestive track or a pipe. It may have the hardware parts to carry out some repair work, but not enough memory to store the code. As it is exploring the surface, when it encounters a certain blemish or problem on the surface, it can download code for that particular repair work, i.e. to control the parts in order to eliminate that problem.

Web services are essential for providing access to certain resources. For example, access to a printer in an Internet café can be provided through a printing web service. Clients in the vicinity can dynamically discover the printing service and use it if the user wants to print documents to that printer.

In fact, web services and facets are not mutually exclusive. They can be used to complement each other. A client can download a facet which accesses a web service. Such as the printing web service described above, a client can download a facet to the service. This way, we can have the best of both worlds.

8.2 TRANSPARENCY OF ADAPTATION

In the facet model, programmers are kept relatively shielded from the details of run-time adaptation. Programmers only need to specify the functionality they require, and they can safely assume that they will receive a facet with resource characteristics suitable for the current execution environment. The amount of resource an application uses is, to a large extent, under the control of the resource manager and the proxy. The resource manager determines how much resource to allocate to a particular application (or functionality), and the proxy finds a facet which can conform to those constraints.

In many systems, the amount of resources an application uses is under the control of the application itself, i.e. the application has in-built mechanisms to change its own run-time resource usage requirements. There is a system wide resource manager which keeps track of

the resources being used, but its role differs from system to system. There are two major approaches. On one hand, the system resource manager is just a source of information. Applications query the resource manager to determine the current resource status, and then adapt themselves accordingly, i.e. they have their own adaptation policy. On the other hand, instead of just keeping track of resource usages, the resource managers also determine the adaptation policy. They carry out resource allocation among applications and inform the applications the amount of resources that have been allocated to them. The applications then adapt their resource usage according to that.

It can be seen that, for both approaches discussed above, applications manage their own resource usage. Application programmers must include mechanisms of adaptation in their programs. In other words, programmers are not only exposed to the mechanism of adaptations but need to know it thoroughly and write it as well. How well an application adapts at run-time, thus, is highly dependent on the skills of the developer.

Yes, in the facet model, application developers do need to build different facets with the same functionality with different resource characteristic. However, they are spared from coding mechanisms for responding to changes in run-time resource availability.

The main advantage of having this transparency is that it makes it a lot easier to build applications. Programming with support for adaptation for a huge variety of devices available can become a burden for programmers. Moreover, applications only possess a local view of the whole system. The resource manager has a global picture and thus is more suitable to do resource allocation and adaptation. It has a centralized control over the usage of the resources in the system. A well-written resource manager can make the system and the execution environment generally more stable and friendlier to work in.

Providing programmers with transparency for important system functions is not something new. Frameworks for Enterprise JavaBeans, CORBA, .Net, etc., also give programmers transparency for persistence, life-cycle management, transactions, etc, making life more easier for programmers and the systems implemented on them more stable. Transparency for adaptation can be seen as the next step in the same direction.

8.3 THE FACET MODEL AND ADAPTABILITY

In chapter two, we discussed five different types of adaptability commonly exhibited by applications, namely – memory, energy, network, device and context adaptability. In this section, we look at how the facet model can be used to enhance each kind adaptability of a mobile application.

The facet model achieves memory adaptability by bringing in facets which have suitable memory characteristics for the current run-time memory conditions. If a device has less run-time memory available, a more memory efficient facet is brought in and executed. In addition, since functionality is brought in little by little (i.e. facet by facet), this makes it more adaptable. For example, stages of a game are downloaded as you go along, saving the memory required for the whole game application.

The facet model is not very suitable for energy adaptability. Facets are just parts of application code. Power-management is usually the responsibility of the operating system, or the middleware, and hence the facet model does not greatly impact energy adaptability. Energy adaptability is best achieved by energy adaptation techniques, such as automatic power off, or dynamic voltage scaling, etc.

Network adaptability is often achieved by network-level adaptation and data adaptation techniques, i.e., by using different protocols, or by changing the quality of data accessed. Network traffic of an application is largely affected by the amount of data it needs to access. Thus, in order to achieve network adaptability, facets with different algorithms, which require less network interaction or access less amount of data, should be used instead.

The facet model also enhances the device adaptability of an application. Different devices have different configurations, e.g. input devices, output devices, processing powers, etc. The facet model makes it easier to adapt the presentation format, UI and even the application functionality. For example, accessing an HTML web page on a notebook could be pretty straightforward. It would require a simple HTML parser facet and associated presentation. However, accessing the same web page on a mobile phone would perhaps require a facet which carries out HTML-to-WML translation in order to adapt the presentation of the page to the display screen of the phone.

Context encompasses a lot of factors – location, time, preferences, surrounding entities. Context adaptability can be easily achieved by using facets suitable, or specially designed for the current context. For example, a client can adapt at run-time to the surrounding entities and be able to work with them. Let's say when a client device detects a nearby printer, it uses a facet, which it receives from the printer itself or from a nearby proxy, enabling it communicate with and to print to that particular printer. With the facet model, applications can also adapt according to the input data they are presented with. Take an MMS (Multimedia Messaging System) application as an example. The type of decoding facet used depends on the type of message that is received. For a jpeg message, a jpeg decoding facet can be downloaded and used. For an mpeg message, an mpeg decoding is used. The advantage of using the facet model for the MMS application is that does not put any limitation on the types of messages the MMS application can decode. As new messaging formats evolve, the MMS application just has to download the suitable decoding facet for that message.

As it can be seen from above, the facet model has a large impact on the degree of memory, device and context adaptability an application possesses.

8.4 CONTEXT AWARENESS

Context-awareness implies being able to detect the context an application is running in and modify the execution of the application according to that. The term context-awareness is often used synonymously with device, network and context adaptability.

In the facet model, context awareness implies choosing the facet suitable for current context. Context detection is carried out by both the client device and the proxy. The client device detects the local context such as time, locale, and the surrounding context such as nearby entities, and network conditions, etc. The proxy keeps track of other context information such as location, surrounding proxies and clients and user preferences, information from surrounding clients, etc. The decision on how to respond to the context is made by the proxy. When a client sends request to the proxy, it includes all the context information it has gathered. The proxy receives this information and combines it with the information it has gathered. It decides which facet will be best for the current context by analyzing and comparing the context information and the shadows of the facets. Thus, both proxies and clients play a part in achieving context awareness.

8.5 DEFICIENCIES OF THE FACET MODEL

As seen in the previous chapter, the facet model has been successfully employed to build applications. However, during the process, a deficiency that we came across was the lack of control or feedback mechanism for facets. In other words, there is no way we can control or enquire about the progress of a functionality until and unless the facet returns. Take, for example, the blurring of an image. Blurring requires a lot of matrix operations and is very computation intensive. When a user invokes a blur operation, he can only know whether the operation was successful when the blur facet returns. While the user is waiting for the results, which may be a long time if the operation was invoked on a slow machine, there is no way of knowing whether the facet is executing, or if it has just hung. However, such a scenario is also present in some of the common software systems currently used.

An application domain that would take a lot of effort to switch to the facet model is that of highly interactive games. For interactive games, functionality, UI and data are very closely linked. Object-oriented programming often makes things easier in this domain. For instance, an enemy can be implemented as an object which contains the enemy's location, code of its movement strategy, code to make it kick and jump, and its related UI. Thus, every entity in the game can be cleanly represented as an object. However, if such a game were to be implemented according to the facet model, functionality, UI and data would need to be more clear-cut. It is all right if strategies, such as those of enemy movement, or for a chess game, are implemented as facets. These strategies can be replaced easily when moving from one stage to the next, i.e. using a facet which implements a more difficult strategy. However, often in games, user interaction invokes rather miniscule functionality, for example, moving forward, kicking, shooting etc. This is not to say that the facet model cannot be used to develop games. The facet model just provides a different style of programming. The game must be carefully designed with the facet model in mind.

8.6 APPLICABILITY OF THE FACET MODEL

Since functionality is brought in as it is required, one of the advantages of the facet model is that it allows devices to run applications which normally would be too large to fit into the device. It helps keep the run-time program size small. The facet model would be the ideal choice for applications with lots of functions, or add-on plugins and filters, like image editors, word processors, etc.

Because of its innate support for adaptation, the facet model is particularly suited for mobile applications, i.e. applications which can move from one device to another, either on their own, or to cater for user mobility. As the user moves from one device to another, the facet model enables the application for state migration and to adapt to the corresponding device.

The domain of scientific simulation and artificial intelligence can also benefit from the facet model. The strategies or the algorithms are implemented as facets. A facet, which contains a strategy, can be easily replaced by another compatible facet with a different strategy. Therefore, it adds flexibility and dynamics to programs.

In fact, the facet model has advantages in the field of grid computing as well. Grids, by definition, are heterogeneous. They aim to make use of the idle cycles of the CPUs, or resources of their constituent nodes. Without functionality adaptation, a machine has to have sufficient "idle" resources before it can be used in grid computation. With the facet model, the grid application could adapt to whatever resources are available in the nodes of the grid, of course, depending on the versions of facets available. This implies that, even if at present a node has less idle resources, those resources can still be put to use. This is very beneficial for the grid because it leads to a better overall usage of the computing resources in the grid and, thereby, improves performance.

Even though in this thesis, the facet model has been discussed in the context of mobile computing, it can be applied to almost any domain. The facet model advocates computing in small. It enables resources, especially memory, to be used efficiently and, thereby, improving performance. Even standalone applications in PCs would benefit from being able to unload an application component once it is no longer used. Thus, the facet model actually has far-reaching benefits.

8.7 SECURITY ISSUE

One important issue which has not been addressed in the current model is security. Facets are located on the public domain and are sent over various insecure public networks to client devices. Security of facets entails making sure that the facet received on the client device is executable and runs as they were meant to, in a safe manner.

Security of facets can be considered as consisting of five aspects

- *Facet Authentication*

  It also involves verifying the information in the shadow of the facet, i.e. confirming that the facet comes from the vendor it claims to be from, confirming the resource usage, etc.

- *Facet Integrity*

  This involves ensuring the facet itself is not malicious. Facets can be written by almost anyone and uploaded on facet servers. And because of the transparency of the model, it becomes crucial to ensure that facets themselves do not carry out any vicious things and come from trusted sources.

- *Facet Validity*

  This means making sure that the facet has not been corrupted or tampered with. Since facets may be sent over insecure networks, it is possible that it may have been tampered by hackers. We need to ensure that the facet is intact before it is executed.

- *Execution Error Logging*

  There may be some facets which often cause clients to hang. In such cases, having a logging system which notifies the proxy of the execution state will be beneficial. The proxy can then screen out facets which are particularly buggy, or have poor performance.

- *Private Facets*

  In the current model, all facets are available to everyone. However, there may be a need for private facets, which have restricted use. These may be special high-security facets which can be used by those only with the appropriate access right.

Further study needs to be carried out on how to make the facet model more secure. Mechanisms which can be explored include encryption, digital signatures, check sums, checking facets thoroughly before including them in the proxy facet lists, using a security model, such as the sand box model in order to protect clients against malicious code, allowing run-time execution logging and sending reports to proxies from time to time, etc.

8.8 SPARKLE ARCHITECTURE

The Sparkle architecture, which has been described in detail in Chapter 4, provides extensive support for the facet model. Every entity in the architecture has a direct role to play with regards to facets. Clients request for and run facets, facet servers store facets, proxies match appropriate facets for clients and execution surrogates execute delegated facets. Since facets are embodiments of functionality, it can be said that the Sparkle architecture has a very strong functionality model.

Running an application requires the definition of three main models

- *The functionality access model*. How application logic is retrieved and from where.
- *The data access model*. Where the user data is located and how it is retrieved.
- *The external resource access model*. How external resources are accessed.

In traditional computing, both functionality and data are assumed to be located locally and can be easily accessed by the use of a filename. External resources, such as printers, are assumed to be connected to the local machine, or available on the local network.

For mobile computing, in which user mobility, resource limitations, network disconnections and instability are prevalent, more robust models need to be applied, so as to ensure the continuity of computation. The Sparkle architecture provides a functionality access model, however, the data access model and the external resource model are not yet well developed.

The Sparkle architecture allows users to access functionality from whichever location they are at, and whatever device they are using. It also provides for the functionality to be adapted to the run-time execution environment.

When a user moves from one device to another, he would expect his data, such as his files, preferences, to be accessible on the new device as well. At present, the Sparkle architecture does not support data migration. It assumes all user data is located in the local device. To make it more suitable for mobile computing, approaches such as mobile file systems, or having a centralized server to store user data need to be investigated and appropriate ones adopted.

A handheld device is considerably different from a PC, especially when it comes to accessing external resources such as printers and scanners. A PC may be directly connected to a printer,

whereas a handheld would have no connections any such resources. Applications may, at times, require access to such resources, thus the devices must be able to make wireless links to them. In the mobile computing context, clients need to be able to locate such resources in close vicinity and be able to access them. Several clients could be competing for access to the devices at a particular instant in time. Subsequently, a resolution mechanism is also required. The Sparkle architecture, at present, does not address this issue. A mechanism commonly adopted is for these external resources to expose themselves as services. Clients, then, locate the services they need, and lease them for some time, i.e. a service discovery and leasing mechanism. This may have an effect on the programming style. The programmer could be aware of these details or it could be hidden from him. The feasibility of this approach and others needs to be investigated thoroughly, and a suitable external resource access model incorporated.

8.9 SUMMARY

In this chapter, we took a step back and looked at several issues regarding the overall picture of the Sparkle system. We looked at how the facet model stands in face of web services. Web services alleviate the problem of functionality adaptation since most of the application logic is executed on servers. However, intermediary proxies are required to carry out data adaptation of the results. The fundamental difference between web services and the facet model is that web services require the movement of data from the client to the service, whereas the facet model moves the code to the client. The direct implications of the previous statements are that

- Code size is often smaller than data size, which may be better for transmission over slow networks.
- In some cases, it may not be possible to move the data, due to privacy and other reasons.
- The service provider has a huge burden of maintaining service availability at all times.
- Web services follow the client-server approach, and hence may not allow for direct peer-to-peer communication which is essential for mobile computing.
- There are cases in which application logic must be executed locally and cannot be carried out remotely.
- However, web services are essential to provide access to external resources such as printers.

One cannot argue which is better than the other. Both web services and facets can be used in order to reap the benefits of both models.

We also looked at some of the issues relating to the facet model, namely the transparency issue, the deficiencies of the facet model and its applicability. In many systems, resource adaptation is under the control of the application itself, i.e. the programmer has to write all mechanisms to respond to change. Our model takes a totally different approach. All the mechanisms are kept transparent from the application designer. The main advantage is that it is much more easier to build applications. Designers just need to provide different versions of facets, and the rest is handled by the system. In addition, the resource manager is probably in the best position to carry out the adaptation policies since it has centralized control over the resources.

We also analyzed how effectively the facet model can enhance the different types of adaptability of an application. The facet model is particularly applicable for memory, device and context adaptability. It can help achieve network adaptability to a limited extent. However, it plays a minimal role in achieving energy adaptability.

Context-awareness in the facet model is achieved by using a facet suitable for the current context. Both client devices and proxies gather context information. When the client makes a request for a facet, it sends the context information to the proxy. The proxy compiles all this information and decides which facet to return to the client. In other words, the proxy makes the ultimate decision on how to respond to a particular contextual change.

The facet model has some deficiencies. From a programmer's point of view, there is no mechanism to find out the progress of the facet while it is being executed. We can only know whether it was successful or not when it returns. In addition, since there is such a marked separation of UI from functionality and data, applying the facet model applications with a lot of user interaction, such as action games, may be cumbersome.

The facet model, however, is suitable for applications in the scientific simulation and artificial intelligence domains. It provides dynamics to the programs because of the flexibility that a facet can be replaced by another facet. Also, the facet is suitable for applications that require functionality adaptation, for example, for applications which need to migrate from one device to another. Grid computing can also benefit from the facet model since it enables applications

to adapt to the idle resources available in the nodes, rather than waiting for the nodes to have sufficient idle resources before they can be utilized.

Finally, we looked at the deficiencies of the Sparkle architecture. The Sparkle architecture has a very solid functionality access model, however it does have a data access model or an external resource access model suitable for mobile computing. At present, there is no support for users to access their data from any device, nor for clients to discover and access nearby resources, such as printers and scanners. Approaches such as mobile file systems, centralized data servers, service discovery and leasing need to be studied and suitable mechanisms need to be incorporated in the Sparkle architecture to make it more apt for mobile computing.

Chapter 9

Conclusion

The whole dissertation is based on employing dynamic component composition to achieve functionality adaptation in mobile computing. This chapter summarizes all the previous discussions and presents avenues for future work.

9.1 Summary and Contributions

Mobile systems are characterized by variation and change. Thus, systems targeted for such a dynamic environment have to cater for the incessant variation. They have to be able to detect run-time changes and adapt to them appropriately.

We looked at the different types of adaptability systems demonstrate in order to accommodate changes. Out of all the adaptation techniques, functionality adaptation is probably the most versatile. Unfortunately, current approaches to functionality adaptation are rather limited and inflexible. They place undue burden on the application programmer, resulting in a larger application size and, not to mention, a limited adaptive capability, which cannot be dynamically extended. Hence, there is a need for a flexible and dynamic functionality adaptation technique which can overcome the above drawbacks. My dissertation aims to fulfill this need.

We proposed dynamic component composition as a means of achieving functionality adaptation. Dynamic component composition can be summarized as follows. Software and applications are made up of components, which are assembled at run-time as they are required. There may be several components carrying out the same task. Which component is used for that particular task depends on the run-time execution environment. Each of these components may have different run-time characteristics and, thereby, achieve functionality adaptation by adapting the execution of the task at hand.

This approach has several advantages in the context of mobile computing, other than functionality adaptation. Since components are brought in when needed and discarded after use, the functionality a device can provide is not limited by its configuration. There is increased scope for peer-to-peer co-operation. Peers can share components, in addition to sharing data and files. Dynamic component composition also supports user mobility and migration adaptation. Since components are located on the network, they can be accessed from any device. When the execution moves from one device to another, the same functionalities can be brought in but with components suitable for the new device, i.e. functionally adapted to the new device.

A new component model – the facet model, was designed especially with dynamic component composition in mind, and an Internet-enabled architecture – the Sparkle system was built to support the facet model. Functionality adaptation, in this case, is achieved by choosing the appropriate component among different ones which have the same functionality.

The main foci of my work are the definition of the facet model and the demonstration of its feasibility. Facets can be considered as just another type of software components, which embody functionality. In addition to the four characteristics of components – being independent, units of composition, providing functionality via well-defined interfaces and having no persistent state, facets only have a single publicly callable method and have no residual state. This allows them to be small and throwable.

The facet model relies on the separation of functionality from data and user interface. Facets are the embodiments of functionality, where as the data and the user interface is stored in the container. It can be said that for an application, the application logic is distributed among the various facets, whereas the state and the user interface are centralized in the container.

Dynamic component systems have been used before, no doubt, but mainly in the area of reconfiguration of long-running systems. For mobile systems, one often hears about service components. However, it is difficult to find a dynamic component system specially addressing adaptation with as much flexibility as the facet model.

In short, we can conclude that the main contributions of this dissertation are as follows:

- It provides a classification of the different types of adaptabilities mobile systems and applications exhibit and also another of the adaptation techniques employed.
- It introduces functionality adaptation and its importance to mobile computing.

- It proposes dynamic component composition as a means of achieving functionality adaptation.
- Most importantly, it defines the facet component model, designed for dynamic component composition.
- It illustrates the facet model's feasibility and applicability in a mobile environment via the Sparkle architecture.

9.2 FUTURE WORK

The current status of the design and implementation does illustrate the importance of the facet model in achieving functionality adaptation for mobile computing. However, it cannot be considered as a full-fledged mobile system. There are several issues which must be further studied and implemented, which include

- *The Facet Model*

  The facet model has proved sufficient for applications which have been implemented using it. However, more work needs to be carried out to demonstrate its feasibility for general applications. It must be experimented with various kinds of applications in order to study its applicability and to provide basis for refinement.

- *Resource Management*

  The current implementation of the resource management in the client system is rather simple. It is restricted by the resource information available to it. At present, it only receives information about heap usage. A native resource monitor, which can effectively keep track of a lot more types of resources, such as memory, power, network, needs to be incorporated. In addition, different adaptation policies need to be implemented and tested in order to determine which would be most suitable under different conditions. Perhaps a priority scheme can also be incorporated.

- *Discovery Mechanism and Peer-to-Peer Interaction*

  At present, there are no discovery or peer-to-peer interaction mechanisms incorporated in the client system. These are essential parts of any system targeted for mobile computing. Mechanisms for the discovery of services, peers and other network entities need to be studied and incorporated. In addition, how peers interact and to which extent adaptation is required needs to be determined.

- *Performance Enhancement*

  The main bottleneck in performance is the delay in the transmission of facets. Several approaches for improvement have been employed, however, there are some avenues which have not yet been explored, for example pushing facets to the client.

- *Facet Security*

  The current model does not incorporate mechanisms for ensuring the integrity of the facets and their execution. Since facets are distributed over public networks, and together with transparency of the model, makes security a crucial issue. Further study on the feasibility of mechanisms such encryption, digital signatures, check sums, sandbox model and execution feedback need to be explored.

- *The Sparkle Architecture*

  The Sparkle Architecture does not possess a clear and strong data access model nor an external resource access model. These models are essential for many kinds of applications. Approaches, such as mobile file systems, centralized data servers, service discovery and leasing, need to be studied and suitable mechanisms need to be incorporated.

With this, we conclude this dissertation. The facet model lays a foundation for a flexible software system for mobile computing. However, there are many issues which need to be resolved to realize a practical system. The facet model forms the ground work on which other research can be based on. Thus, its significance and contribution cannot be undermined.

BIBLIOGRAPHY

1 E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. *In Proceedings of* ACM SIGCOMM,pp. 178-189, September 1998.

2 D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. *In Proceedings of 4th Symposium on Operating Systems Design and Implementation*, pp. 213-226, October 2000.

3 N. Apte and T. Mehta. *Web Services – a Java Developer's Guide using E-speak.* Prentice-Hall, 2002.

4 L. Bao and J. J. Garcia-Luna-Aceves. A New Approach to Channel Access Scheduling for Ad Hoc Networks. *In Proceedings of the 7th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '01)* pp. 210-221, July 2001.

5 G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. B. Sussman, D. Zukowski. Challenges: an Application Model for Pervasive Computing. *In Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking(MOBICOM '00)* pp. 266-274, August 2000.

6 P. Brereton and D. Budgen. Component-based Systems: a Classification of Issues. *In IEEE Computer, vol. 33 (11),* pp. 54-62, November 2000

7 G. Cao. A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments. *In Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '00),* pp. 200-209, August 2000.

8 Carnegie Mellon Software Engineering Institute. Component-Based Software Development / COTS Integration. http://www.sei.cmu.edu/str/descriptions/cbsd_body.html

9 Carnegie Mellon Software Engineering Institute. COTS and Open Systems – An Overivew. http://www.sei.cmu.edu/str/descriptions/cots_body.html

10 P. Castro, P. Chiu, T. Kremenek and R. R. Muntz. A Probabilistic Room Location Service for Wireless Networked Environments. *In Proceedings of the 3rd International Conference in Ubiquitous Computing* (*Ubicomp '01),* pp. 18-34, September 2001.

11 Cetus Links. http://www.cetus-links.org

12 S. Chandra, C. S. Ellis, and A. Vahdat. Mulitmedia Web Services for Mobile Clients Using Quality Aware Transcoding. *In Proceedings of the Second ACM International Workshop on Wireless Mobile Multimedia(WOWMOM '99),* pp. 99-108, August 1999.

13 G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, November 2000

14 I. Chlamtac and J. Redi. Mobile Computing: Challenges and Potential. In *Encyclopedia of Computer Science,* 4th Edition, International Thomson Publishing, 1998.

15 Y. Chow. *A Lightweight Mobile Code System for Pervasive Computing*. Master's Thesis, Department of Computer Science and Information Systems, The University of Hong Kong, August 2002

16 I. Clarke, O. Sandberg, B. Wiley, T. W. Hong. Freenet: A Distributed Anonymous Information Storage

and Retrieval System. *In Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability,* pp.46-66, July 2000

17 COM. http://www.microsoft.com/com/

18 A. Corradi, R. Montanari and C. Stefanelli. How to Support Adaptive Mobile Applications. *In Proceedings of WOA2001, September 2001*

19 P.T. Cox, B. Song, A Formal Model for Component-Based Software, *In Proceedings of IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering,* September 2001.

20 Dom4j, The flexible XML Framework for Java. http://www.dom4j.org/

21 D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML – the Catalysis Approach*, Addison-Wesley, 1997.

22 C. Efstratiou, K. Cheverst, N. Davies and A. Friday. An Architecture for the Effective Support of Adaptive Context-Aware Applications. *In Proceedings of the Second International Conference on Mobile Data Management (MDM'01)*, pp. 15-26, January 2001.

23 J. Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. *In Proceedings of the 17th ACM Symposium on Operating System Principles*, pp. 48-63, December 1999.

24 A. Fox, S. Gribble, Y. Chawathe and E. A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *In IEEE Personal Communications*, Special Issue on Adaptation, August 1998.

25 A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *In IEEE Transactions on Software* Engineering vol. 24(5), pp. 342-361,1998.

26 J. Gao, P. Steenkiste, E. Takahashi and A. Fisher. A Programmable Router Architecture Supporting Control Plane Extensibility. *In IEEE Communications Magazine, Special Issue on Active, Programmable, and Mobile Code Networking, vol. 38(3)*, pp. 152-159, 2000.

27 D. Garlan and B. R. Schmerl. Component-Based Software Engineering in a Pervasive Computing Environment. *In Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering,* May 2001.

28 Gnutella. http://www.gnutella.com

29 S. D. Gribble, M. Welsh, J. R. von Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross and B. Y. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *In Computer Networks vol. 35(4),*pp. 473-497, 2001.

30 R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A System Architecture for Pervasive Computing. *In Proceedings of the 9th ACM SIGOPS European Workshop*, pp. 177-182, September 2000.

31 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. *In Proceedings of the 8th Workshop on Hot Topics in Operating Sy*stems, pp. 128-132, May 2001.

32 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.

33 T. Gross, P. Steenkiste and J. Subhlok. Adaptive Distributed Applications

on Heterogeneous Networks. *In Proceedings of the 8th Heterogeneous Computing Workshop*, pp.209-218, April 1999.

34 R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret and J. Rubas. Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing. *In IEEE Personal Communication, vol. 5(6),* pp. 8-17, 1998.

35 A. Harter, A. Hopper, P. Steggles, A. Ward and P. Webster. The Anatomy of a Context-Aware Application. *In Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '99)* pp. 59-68, August 1999.

36 W. R. Heinselman, J. Kulik and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. *In Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking(MOBICOM '99)* pp. 174-185, August 1999.

37 G. D. Holland, N. H. Vaidya, and P. Bahl. A rate-adaptive MAC protocol for multi-Hop wireless networks. *In Proceedings of the 7th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '01)* pp. 236-251, July 2001.

38 HP Web Services Platform. http://www.hpmiddleware.com/SaIsapi.dll/SaServletEngine.class/products/hp_web_services/default.jsp

39 A. C. Huang, B. C. Ling and S. Ponnekanti. Pervasive Computing: What is it Good for? *In Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pp. 84-91, August 1999.

40 Image Processing and Analysis in Java. http://rsb.info.nih.gov/ij/

41 IP Routing for Wireless/Mobile Hosts. http://www.ietf.org/html.charters/mobileip-charter.html

42 J-consortium JEFF. http://www.j-consortium.org/jeffwg/index.shtml

43 Java-Linux. http://www.blackdown.org/

44 JINI Network Technology. http://wwws.sun.com/software/jini/

45 D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. *In Mobile Computing*, ed. T. Imielinski and H. Korth, Kluwer Academic Publishers, 1996.

46 V. Kanodia, C. Li, A. Sabharwal, B. Sadeghi, and E. W. Knightly. Distributed Multi-Hop Scheduling and Medium Access with Delay and Throughput Constraints. *In Proceedings of the 7th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '01)* pp. 200-209, July 2001.

47 F. Kon, R. H. Campbell, M. D. Mickunas, K. Nahrstedt and F. J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. *In Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00*), pp. 201-210, August 2000.

48 F. Kon and T. Yamane. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. *In Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001),* pp. 15-30, February 2001.

49 J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A case study. *In Proceedings of the 4th International Conference on Configurable Distributed Systems*, pp. 91-100, 1998.

50 R. Kravets and P. Krishnan. Application-driven Power Management for Mobile Communication. *In Wireless Networks (WINET), vol.6(4),* pp. 236-277, 2000.

51 T. Kunz and J. P. Black. An Architecture for Adaptive Mobile Applications. In *Proceedings of the 11th International Conference on Wireless Communications (Wireless '99)* pp. 27-38, July 1999.

52 W. M. Kwan. *A Distributed Proxy System for Functionality Adaptation in Pervasive Computing Environments*. Master's Thesis, Department of Computer Science and Information Systems, The University of Hong Kong, August 2002.

53 Y.W. Lee, K. S. Leung, and M. Satyanarayanan. Operation Shipping for Mobile File Systems. *In the Proceedings of the 1999 USENIX Annual Technical Conference,* June 1999.

54 Q. Li, J. Aslam, and D. Rus. Online Power-Aware Routing in Wireless Ad-hoc Networks. *In Proceedings of the 7th ACM/IEEE International Conference on Mobile Computing and Networking(MOBICOM '01)* pp. 97-107, July 2001.

55 J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger and R. Morris. A Scalable Location Service for Geographic Ad Hoc Routing. *In Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '00),* pp. 120-130, August 2000.

56 Q. Li and D. Rus. Sending Messages to Mobile Users in Disconnected Ad-hoc Wireless Networks. *In Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '00)* pp. 44-55, August 2000.

57 R. Litiu and A. Prakash. DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. *In the Operating Systems Review, vol35(2),* 31-42, April 2001

58 Microsoft Developer's Network, http://msdn.microsoft.com

59 Micorsoft .NET. http://www.microsoft.com/net/

60 Napster. http://www.napster.com

61 .Net Assemblies and Manifest. http://www.dotnetextreme.com/articles/assemblies.asp

62 B. Noble and M. Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. *In Mobile Networks and Applications (MONET), vol. 4(4),* pp. 245-254, 1999.

63 Object Management Group. http://www.omg.org/

64 OMG CORBA. http://www.corba.org/

65 Oracle 9i Dynamic Services, http://otn.oracle.com/tech/webservices/content.html

66 Pervasive Computing. http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci759337,00.html

67 P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *In Proceedings of the 18th ACM Symposium on Operating System Principles*, pp. 89-102, October 2001.

68 F. Plasil, D. Balek, R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. *In Proceedings of the International Conference on Configurable Distributed Systems (ICCDS'98),* May 1998.

69 B. Raman, S. Agarwal, Y. Chen, M. Caesar, W. Cui, P. Johansson, K. Lai, T. Lavian, S. Machiraju, Z. M. Mao, G. Porter, T. Roscoe, M. Seshadri, J. Shih, K. Sklower, L. Subramanian, T. Suzuki, S. Zhuang, A. D. Joseph, R. H. Katz and I. Stoica. The SAHARA Model for Service Composition Across Multiple Providers. *In Proceedings of the International Conference on Pervasive Computing,* August 2002.

70 R. Rock-Evans. *DCOM Explained,* Digital Press, 1998.

71 A. Rofail and Y. Shohoud. *Mastering COM and COM+*. Sybex, 2000.

72 E. Roman. *Mastering Enterprise JavaBeans and the Javea 2 Platform, Enterprise Edition.* Wiley Computer Publishing, 1999.

73 E. M. Royer and C.-K. Toh. A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks. *In IEEE Personal Communications Magazine*, pp. 46-55, April 1999.

74 T. Schiepek. A Comparison of Component Technologies. *In Seminar on Component-Based Software Development.* Germany, 2000.

75 S. K. Shrivastava and S. M. Wheater. Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications. *In Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, May 1998.

76 J. Siegel. *CORBA 3 – Fundamentals and Programming, 2nd Edition.* OMG Press, 2000.

77 T. Simunic, L. Benini, P. W. Glynn and G. D. Micheli. Dynamic Power Management for Portable Systems. *In Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '00)*, pp. 11-19, August 2000.

78 P. Sridharan. *JavaBeans – Developer's Resource.* Prentice Hall, 1997.

79 I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H.Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *In Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM 2001)*, pp.149-160, August 2001.

80 Sun Open Net Environment, http://wwws.sun.com/software/sunone/

81 SyncML, http://www.syncml.org/

82 C. Szyperski. *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

83 C. Szyperski. Components and Objects Together. *In Software Development Online.* http://www.sdmagazine.com, May 1999.

84 N. Vaidya, P. Bahl and S. Gupta. Distributed Fair Scheduling in a Wireless LAN. *In Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '00)*, pp. 167-178, August 2000.

85 Visual Studio .Net Training Tour, DOTNET, 2001

86 WebSphere Application Server. http://www-3.ibm.com/software/webservers/appserv/

87 The Xerox PARCTab, http://www.ubiq.com/parctab/

88 Y. Xu, J. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. *In Proceedings of the 7th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '01)* pp. 70-84, July 2001.

89 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*, pp. 156-163, March 2000.

90 A. B. Zaslavsky and Tari. Mobile Computing: Overview and Current Status. *In Australian Computer Journal, vol.30(2)*, pp. 42-52, 1998.

91 B. Zenel and D. Duchamp. A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment. *In Proceedings of the 3th ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '97)*, pp. 248-259, September 1997.