A Distributed Proxy System for Functionality Adaptation in Pervasive Computing Environments

by

Kwan Vivien Joanna Wai Man

Temporary Binding for Examination Purposes

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 Distributed Proxy System for Functionality Adaptation in Pervasive Computing Environments"

submitted by

Kwan Vivien Joanna Wai Man

for the degree of Master of Philosophy

at the University of Hong Kong

in August 2002

Pervasive computing, with its distinctive feature of computation at anytime, anywhere, and on any device, has attracted a lot of attention in the last decade. The heterogeneity of client devices, limited capabilities of small devices, and users' high mobility together pose a great challenge to the provision of web contents and services to the clients. They can be made more suitable to the clients via some form of adaptation by taking into account the clients' execution contexts. Content adaptation is an existing solution in adapting web contents to resource-constrained devices. Web contents are usually transcoded such that they can fit into these devices. For services, the recent emergence of Web Services tries to take care of the distinctive features of pervasive computing by using dedicated servers for providing services to the clients. However, this approach cannot fit the needs of all the clients. There might be cases where some clients preferred to have the services provided in the client devices. In that case, another approach is to adapt codes to the clients so that they can be executed in the client devices. Although content adaptation is a well-established area of research, very little has been done on the adaptation of program or service codes in the existing literature. The adaptation of services, called functionality adaptation in this thesis, is to adapt service codes such that they can provide the desired functionality in the client devices.

In this thesis, we present a proxy-based approach for functionality adaptation. The greatest challenge in adapting a service to a resource-constrained device is to estimate the resource usage required to provide the desired service, which is the total resource usage required by the service codes for the execution. This resource usage is not a static value that can be known before returning the service codes to the clients, but a dynamic value that depends on the execution and cannot be known until run-time. This variation in the dynamic resource usage makes it difficult for the proxy servers

to determine whether a functionality can be completed in the client device. This thesis presents a conservative solution for supporting functionality adaptation in pervasive computing environments.

As a proof-of-concept for functionality adaptation, a simple prototype of the proxy server has been implemented and tested. Experiments show that the adaptation quality is quite good, and can achieve a quality of over 65% when compared to providing the functionality in the ideal case. Apart from the quality, an average processing time of 300ms also suggests that the proxy server is of a reasonable performance and is suitable to be used in pervasive computing environments. Comparison is also done with random facet selection and the results indicate that although functionality adaptation can only have half the performance, a quality of 1.6 times better can be achieved. Functionality adaptation has sacrificed the performance for a better quality, which is more important to the clients if the performance is reasonable. (487 words)

Kwan Vivien Joanna Wai Man

August 2002

A Distributed Proxy System for Functionality Adaptation in Pervasive Computing Environments

by

Kwan Vivien Joanna Wai Man

A thesis submitted in partial fulfillment
of the requirements for
the degree of Master of Philosophy
at the University of Hong Kong.

August 2002

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 qualification.

Kwan Vivien Joanna Wai Man

August 2002

Acknowledgements

I would like to thank my supervisors, Dr. Cho-Li Wang and Dr. Francis Chi-Moon Lau, for their valuable guidance and support. They have been patient with me and my project group members, and provided precious comments in our papers and thesis. I would also like to thank my project members, Miss Nalini Moti Belaramani and Mr. Chow Yuk, for their valuable discussions and comments on my research. They have also stimulated me in bringing up the important issues in my research. Thanks should be given to Mr. Harris C. H. Chiu for his effort in finding applications to demonstrate the features of our Sparkle Project. Special thanks should be given to Dr. Joe K. W. Chong and Dr. S. M. Yiu for their encouragements and support in these two years. I have really learnt a lot from them and it is fun working with them as well. They have given me many enjoyable hours.

Thanks should also go to my friends, Mr. Vincent W. Y. Lum, Mr. L. L. Cheng, Mr. Felix K. M. Cheung, Mr. K. F. Kan, and Mr. Reynold C. K. Cheng, not only for giving me an enjoyable life in my postgraduate studies, but also their help and support in my research; Dr. Isaac K. K. To in sharing his Linux knowledge and experiences, especially the use of Emacs macros that is very useful in speeding up my testing and evaluation process; all members of the Linux Interest Group and Linux Support Group who have shared their happiness and their experiences that make me enjoy my postgraduate life; as well as all the Systems Research Group members for sharing their researches in the weekly research meetings. My room-mates and ex-room-mates should not be missed in the acknowledgement for providing me a nice environment for doing my research.

Last but not the least, I would like to express my deepest gratitude to my parents, especially my mother, for their continuous support and concern. Their support has given me the energy to work on my research when I was depressed. Thank you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the recent decade, we have witnessed the proliferation of mobile devices and a revolution in people's way of computing. Today, computing is no longer limited to a particular location using some stationary computing devices, such as Personal Computers (PCs) at home or in office. With the advances in the computing technologies, computing can now be done on a heterogeneity of mobile devices, such as laptop computers and information appliances like Personal Digital Assistants (PDAs), pagers, smart phones, etc. Services can be accessed using these mobile devices wherever and whenever they want them: at home, in a meeting, or even when traveling. Computing is no longer restricted to any location or computing device.

Pervasive computing [38] is a term used to describe the kind of computing that can be done anytime, anywhere, and on any device. Computing can be done whenever a computing device is available; e.g. with the PC at home and in office, the PDA you carried during traveling, the laptop brought into the meeting room, or even the thin client inside a coffee shop. All these devices help accessing information or services in one seamless, integrated system [39]. There is no fixed association between a device and a user, meaning that these devices do not need to be owned, and can be any device that is come across. For example, while walking down the street, you suddenly remember that you have forgotten to book an airplane ticket for your boss. Access to the Web for a booking service is urgently needed. You, therefore, walked into a coffee shop around the corner, where a few computing devices are available, and book a ticket with one of the devices. As can be seen, in pervasive computing, users are free to move and compute without the need to carry any device.

By allowing computation to be done anytime, anywhere, and on any device, pervasive comput-

ing environments exhibit characteristics that are not shared by traditional computing environments. These special characteristics include:

- **Heterogeneity of computing devices.** Computing devices ranges from small mobile devices, such as smart phones and PDAs, to tablet PCs, laptops, and stationary devices such as PCs. Each of these devices has a different hardware configuration and capability, and possibly a different software platform for program execution.

- **Limited device capability.** Many mobile devices are designed to be small for easy portability. They are small not only in terms of form factor, but also computing power, memory, display screen size and battery power. All these limit the type of computing that can be done on these devices.

- **Limited network connectivity.** The wireless connection used in the mobile devices is usually bandwidth-limited and unstable when compared with the wired connection. Although new networking standards (such as GPRS and Bluetooth) exists, the comparatively slow speed is always a barrier for running large applications using mobile devices.

- **High mobility.** Users in pervasive computing environments are highly mobile. Without the need to have a fixed association with the computing devices, users can move freely from place to place with or without any mobile device. This high mobility nature also results in a continuous change of the computing environment, or *execution context*, that can affect the execution.

These characteristics are essential to help identifying the requirements needed for pervasive computing.

1.1 Requirements of Pervasive Computing

In order to enable pervasive computing, information or services delivered should be able to satisfy the requirements stemming from the above characteristics. These requirements include:

- **Client-device adaptability.** Client devices of different capabilities have different requirements to the information or services being retrieved. Failure in satisfying these requirements results in the inability of the devices to process the information or services properly. For

example, web pages normally designed for PCs cannot be displayed properly on a PDA without adaptation. Also, a device with limited amount of memory cannot be used for running large monolithic applications (e.g. a full-featured image editor) that consumes a large amount of memory resources and exceeding its capability. In order for the client to process them properly, information or services have to be adapted to the client devices, according to their capabilities.

- **Device-user adaptability.** The free associativity between devices and users allows devices of the same capability (e.g. the same model) to be used by different users. The requirements posed by these clients in accessing the information or services might be different. This is due to the different requirements of the users, which are referred to as the user preferences. Adapting to the device-user according to their preferences is, therefore, needed.

- **User mobility support.** User mobility is the ability of a user to move and compute, while being treated as the same user independent of the physical location. In pervasive computing, users are moving all the time, requesting information or services from different locations. Due to this highly mobile nature of the users, there is a need to minimize the disturbances made to the computing experiences of the users, such as maintaining the same environment (e.g. visited web pages, prepared documents), and providing service continuation for on-going services. All these support for a user at different locations are considered as user mobility support.

- **Context-awareness.** With the ability to compute any place, any time, the execution context of a client changes quite often in pervasive computing environments. The information required, the type and quality of service, etc. are affected by a client's execution context. For example, a detailed road-map when driving in the outback VS a simplified one in the city; benchmarking service used in office VS payment service used at home. Better services can only be provided if information about the clients' execution contexts is known.

In fact, *context-awareness* [1] is the key to the first three requirements: adaptation and user mobility support can only be provided (and best-fit the clients) with information about their execution contexts.

1.2 Existing Approaches for Pervasive Computing

In satisfying the requirements of pervasive computing, many researches [40, 17, 46, 41] have focused on the adapting of information to the client devices and the device users. Content adaptation, which is concerned with the adaptation of data content in data servers to small devices, is a very active research area [23, 40, 27]. Content adaptation mechanisms offer to bridge the gap between the limited device capabilities of small devices and the richness of contents of many online documents. With content adaptation, data can be made suitable for computing devices of any kind. This suits the concept of Web Services [37, 48] for pervasive computing, where services are provided by dedicated servers, and the results of executing the services are returned to the clients. In the world of Web Services, content adaptation is normally required for adapting the resulting contents to the client devices. Microsoft .NET [43], Sun Microsystems Open Net Environment [45], and the University of Washington's One.World [49] are a few examples of architectures and projects where content adaptation can be used in the world of Web Services. By adapting the resulting contents returned by the servers, clients are able to access services of any kind using any computing device that is available, thus enabling pervasive computing.

1.3 Motivation

Information is, no doubt, important in the traditional information web. This is also the case in the emerging Web Services or where "thin clients" are involved. In order to provide a solution for computing in heterogeneous devices, services are provided by dedicated servers instead of leaving them to the clients. As a result, codes to be executed in providing the services are done on the resource-rich servers. There is no need for the clients to execute any service codes. Only data resulting from the provision of services is returned to the clients. Adapting data contents to the clients is, therefore, important in enabling pervasive computing.

However, we believe that only adapting data contents is not enough for pervasive computing. Data only contributes part of the computing. The other part is bring about by codes. Although codes can be executed on resource-rich servers, as in the case of Web Services, a problem exists in this approach where executions are all delegated to the servers. With executions being delegated, data needs to be passed to the servers that provide the services. This is a privacy concern. In some cases, data required to be processed by these services are confidential to the device user. For example, a service is required for converting a business proposal to another document format. It is a bad

idea (and a limitation) that the conversion can only be done on the servers, forcing the confidential document to be passed to the servers. Even if the document is encrypted, the contents are still exposed to the servers when it is decrypted for the conversion. Apart from the privacy issue, data to be processed is, sometimes, quite large to be passed to another entity for processing. In pervasive computing, the client device can be a PC that can store quite a large amount of data. These data might need to be processed, and passing a large amount of data in the network creates a large amount of traffic and is also quite inconvenient.

Another problem in providing services by dedicated servers is that the services cannot move with the clients. They have to be provided by the dedicated servers no matter where and how the clients move. Sometimes, the clients might need a service provided by a server that is quite far away from them. Contacting a distant server for providing a service results in poor user perceived latency. It would be better if the provision of the services can be moved to another machine that is nearer to a client, so that the user perceived latency can be improved.

In light of all these restrictions, another alternative is to allow the conversion to be done in the client device; i.e. allowing executions to be done by the clients. Allowing executions in the client devices is not easy. Similar to data contents, there also exists a gap between small devices' capabilities and the resources needed by many programs. Adapting the codes to a variety of resource-constrained devices is a solution. Therefore, instead of focusing on adapting data contents, we put the emphasis on adapting service codes; i.e. code adaptation. In our research, we are concerning how service codes can be adapted to the client, so that they can be executed in the client devices, enabling pervasive computing.

1.4 Functionality Adaptation

Code, when executed, provides a certain functionality. Different codes achieve different functionalities. Therefore, in adapting service codes, we are, in fact, adapting the functionality to the client. The term *functionality adaptation* is used here for the emphasis of adapting service codes to the client for execution, and providing the desired functionality.

Instead of indicating a specific code to adapt, functionality adaptation specifies the functionality. In that sense, suitable service codes need to be adapted to provide the desired functionalities to the client. This requires:

- the service codes being able to be executed and with their functionalities completed in the

client device.

- the service codes, when executed, provide the functionalities needed by the client
- the service codes have their properties satisfying the requirements of the device user.

Although both content adaptation and functionality adaptation have to adapt to the capabilities of the client devices, adaptation techniques used in content adaptation might not be suitable for functionality adaptation. In content adaptation, transformation (or transcoding) [40, 46] is the most common technique for bridging the gap between the capabilities of small devices and the richness of the contents of online documents. Documents are transformed such that less resources is used while maintaining most of the semantic contents. Compression and color depth reduction are a few examples of transcoding techniques. Despite of its popularity in content adaptation, transcoding operations are lossy, and cannot be used in functionality adaptation. Codes used for execution must be complete, and cannot afford losing any code segment. Otherwise, execution may fail to provide the desired functionality. Even if a lossless transformation is possible, it is still difficult to ensure the functionality is preserved; i.e. given a code $code_A$, there is no proper way to ensure the transformed code $code_B$ can provide the functionality that $code_A$ provides. Therefore, adapting service codes to the capabilities of the client devices requires different versions of service codes that provide the same functionality, each suitable for a different range of device capabilities. Functionality adaptation is then achieved by *selecting* service codes that are able to satisfy the client device's capability during execution.

Although selecting a suitable version of the service code seems to solve the adaptation problem, the main difficulty of functionality adaptation is to determine the resource usage of a piece of service code. Unlike content adaptation, data contents to be adapted have fixed resource usages. They consume a static amount of resources in the client devices, whose values can be known before returning to the client. This helps in deciding the suitability of returning the data contents to the client devices. On the contrary, service codes to be adapted in functionality adaptation have their resource usages vary with the executions. The amount of resources used by executing a piece of service code cannot be known until run-time. This dynamic resource usage depends on the control constructs used in the service codes and, in most cases, the size of input data at run-time. All these dynamisms make the decision of whether a service code is suitable to be executed by a client device tough.

1.5 The Distributed Proxy System

In order to truly enable pervasive computing, the support of functionality adaptation is necessary. Functionality adaptation can be supported in the servers or the intermediary proxies. Having it supported in the servers has the disadvantage of placing additional computational load and resource consumption on the servers. A proxy-based approach, on the other hand, is more flexible, and can reduce the load and resource consumption on the servers. In addition, in supporting everywhere computing, a distributed architecture is usually used.

A distributed proxy system is, therefore, designed for functionality adaptation in pervasive computing environments. It is based on our Sparkle Project, in which an infrastructure has been designed for pervasive computing. The infrastructure, with the use of *facets*, provides the basis for our "ideal" pervasive computing environment. Facets (mobile service code components) are downloaded to the client devices for executing and providing the services.

The proxy system is, in fact, part of our Sparkle Project, which is an infrastructure designed for pervasive computing. It plays an important role in the project by helping the clients in selecting service codes (facets) suitable for the executions. Besides the proxy system, the use of facets that can be dynamically downloaded to the client devices for execution and the use of mobile agents for migration (in cases where the functionality cannot be completed in the client device) are also important parts of the project. They are discussed separately in the Masters theses [2] and [7] respectively.

Apart from the difficulties mentioned earlier, the flexibility of providing a service using facets also introduces another level of dynamism to the adaptation of the service codes to the clients. Service codes are dynamically bound to form a service for flexibility. This dynamic binding of service codes implies that the resource usage of a service cannot be pre-determined statically, which increases the difficulty of the proxy system in adapting a functionality to the client devices.

In order to adapt the functionality to the client, a *two-phase adaptation* technique is used. The first phase adapts the service codes to satisfy the requirements imposed by the client, followed by the second phase that adapts them to the device user. During the adaptation process, there are certain check-points for ensuring the functionality, the resource usage, and the properties of the service codes satisfy the execution contexts of the clients.

In adapting a service code to satisfy the capabilities of a device, prediction of the total resource usage for executing the functionality is needed. This requires predicting the service codes that

constitute a service. With all these predictions, the proxy system is able to *provide a guarantee* to the client as to whether the service codes returned to them are able to provide the desired functionalities in the client device, provided that certain conditions are satisfied. This is a *conservative prediction*, which only serve as a means of providing a guarantee to the client, and not aimed at correctly predicting the service codes that will be dynamically bound for use at run-time. Nevertheless, the prediction can be refined when more and more information about the execution context is known at run-time, so that service codes that are to be used at run-time can be better reflected in the refined prediction.

1.6 Thesis Contribution

The distributed proxy system is the first attempt to have a proxy system designed for functionality (code) adaptation. It helps clients to select suitable service codes that constitute a desired service, and returns them to the clients for execution. These techniques, if co-exists with the content adaptation techniques used in transcoding proxies, enable pervasive computing to be truly supported. In our research, we have identified the following contributions:

1. We have introduced the two-phase adaptation technique in adapting service codes to the client, so that services can be provided by the client devices. Clients do not need to rely on the servers for providing the services, thus protecting their privacies. Being able to adapt service codes to a diversity of client devices also allows computing to be done anytime, anywhere, using any devices.

2. With the intelligence of the proxy system in selecting service codes for the clients, services can be better adapted to the clients' execution contexts, and suitable for the users using the devices. Results show that the service codes selected can achieve a better quality than a random facet selection with the sacrifice of a little bit of the performance.

3. We have proposed a conservative prediction to solve the problem of adapting service codes to the capabilities of client devices. The dynamic resource usage for executing a functionality can be predicted, which helps to provide a guarantee that the service codes being adapted are able to complete its functionality in the device, if certain conditions are met. Once a service starts its execution, it is guaranteed to complete the service in the device.

4. By specifying the functionality instead of a service code for adaptation, service codes have

to be chosen for their abilities to provide the desired functionality. We suggest the use of *capability matching* to match service codes that can provide a greater capability of the functionality. This concept can be extended to match any behaviors (e.g. human behavior) that can be formally described and compared.

1.7 Thesis Organization

This thesis is organized as follows. First, some of the related works are described in Chapter 2. Next, an overview of the Sparkle Project, which is the basis of our work, is presented in Chapter 3. The role of our distributed proxy system is also highlighted in the chapter. Chapter 4 then introduces the functionality adaptation. The needs, requirements, and difficulties of the adaptation is discussed. It also describes the mechanisms that our proxy system uses for achieving the adaptation. Chapter 5 focuses on the resource filtering problem for functionality adaptation. A methodology used for solving the filtering problem is introduced. It also illustrates that the methodology is safe and can be refined for providing better services to the clients. A simple prototype of the distributed proxy system is given in Chapter 6, with its design and implementation shown. Chapter 7 evaluates the prototype for functionality adaptation and its impact to pervasive computing. Chapter 8 concludes the thesis by giving a conclusion and suggesting a list of future works.

Chapter 2

Related Works

Surveys are conducted to compare our work with others. This chapter shows a list of works that share the same or a similar aim as our proxy system — enabling truly pervasive computing by adapting service codes to a diversity of resource-constrained devices, and which best-fit the user and other execution contexts affecting the computation. We believe that fully relying on resource-rich servers for providing services is not the best way to achieve pervasive computing. Componenized services should be able to be executed in the client devices, and adapting service codes for the service components is the key behind this.

The comparison is divided into three areas: the *adaptation strategies* used to overcome the inherent weaknesses of computing in small devices, the techniques used in the *lookup services* for locating services according to the requirements specified by the clients, and other *special features* for supporting pervasive computing.

2.1 Adaptation Strategies

The ability to adapt information and services to a diversity of computing devices is the key to pervasive computing. Adaptation can be performed in the server, the client, or the intermediary proxies. For flexibility, intermediary proxies are normally used in adapting the resulting contents before returning to the clients.

2.1.1 Content Adaptation

Among all the adaptations, content adaptation attracts the most attention [22, 40, 23, 17]. Researchers for pervasive computing mostly acknowledge the concept of Web Services in eliminating

the need for executing codes in resource-constrained clients, and focus on the adaptation of web contents to the client devices. Different approaches of content adaptation can be identified in these efforts [9, 4, 3].

2.1.1.1 TranSend

University of California Berkeley's TranSend [9] is an attempt to adapt web contents to heterogeneous client devices. Adaptation is performed in TranSend's proxies. Documents are generated on-the-fly to meet the client's hardware requirements. To adapt the documents, *distillation* is used to transcode the contents so that they can be handled by the resource-constrained devices. Distillation is a highly lossy, real-time, and data-type specific compression. Contents are distilled intelligently according to their data types, e.g. removing color information and formatting information that the devices are not able to understand. Quality is, thus, sacrificed to preserve most of the semantic contents.

Techniques of distillation is not suitable to be used in adapting service codes. Distillation is a highly lossy operation, meaning that part of the information is lost to reduce the resource consumption while preserving its semantics. Adapting service codes is different from adapting contents. Service codes are for execution and needs to be complete for correct execution, while contents are for presentation and their structures can be syntactically different but with the same semantic contents. It is difficult to sacrifice some information of a code while preserving its functionality. Other approaches for adapting service codes are, therefore, needed.

2.1.1.2 Digestor

Digestor [4] is software system that uses *automatic re-authoring* for adapting online documents. It can be run on HTTP proxies so that online documents can be adapted to different client devices. Re-authoring is the re-structuring of the documents such that they can be presented on the resource-constrained devices. Techniques for re-authoring includes outlining (e.g. section headers made into hyperlinks with the contents elided from the document), first sentence elision (the first sentence of each block made into a hyperlink), and image reduction and elision (images are scaled down and the reduced images have hyperlinks back to their originals). Digestor makes use of an automatic re-authoring system to select the best combination of the re-authoring techniques for the document/display-size pair. The automatic system has a heuristic planner to help making re-authoring decisions, which is based on the heuristics captured in manual re-authorings. The

heuristic planner basically works by expanding a version of the document through application of a single transformation technique, and selects the most promising one (the one that uses the smallest display area). Transformation techniques are then applied to the selected version, and the process repeats until a version is "good enough" to be returned to the client.

The concept of re-authoring requires semantic understanding of the structure of the document, and is difficult to be used in code adaptation. Even if re-coding is possible, it is usually difficult to determine whether a transformed code can perform the functionality provided by the original code. Added to the difficulty is the format of the code. Source codes are required for re-authoring, but service codes for execution are usually deployed in binary format. Moreover, re-authoring also depends on the language used for coding. Codes written in C++ and those written in Java require different ways to re-code, which makes it difficult to be automated in the proxy system. All these problems need to be overcome in order for re-authoring techniques to be suitable for functionality adaptation.

2.1.1.3 Mowser

Mowser [3] is an *active transcoding* proxy that modifies the HTTP stream and change its content in situ. Compared with TranSend, Mowser has greater control of the content that a client receives. It allows users to set their preferences based on their needs and the capabilities of the devices, such as the video resolutions, sound and color capabilities, maximum sizes for each type of files, etc. These preferences are stored in an Apache server. When the proxy receives a request from the client, it looks up the preferences that is stored with the IP address of the client, and modifies the HTTP request according to the preferences before sending to the server. Mowser introduces what is called *content negotiation*. Different variants (or *fidelities*) of the web contents are stored in the server. The proxy asks the server for one of the fidelities that is available, and which satisfies the requirements of the client. If the one that is returned by the server is still too large for the client, transcoding is applied. The kind of transcoding that is applied depends on the file type. For example, the size or color depth of an image is reduced, and representative frames of a video file are selected to convey the same information. The proxy can also parse the HTML stream to remove active contents and tags that the clients cannot handle. Any contents that cannot be processed by the client are not returned.

As aforementioned, transcoding techniques that remove information from the contents cannot be applied to codes. Therefore, our proxy system cannot apply techniques similar to selecting

representative frames (e.g. selecting representative codes) for reducing the code size, as this kind of selection is not meaningful for codes and is likely to change their execution behaviors. Instead, our proxy system behaves like the servers in Mowser, where selection of code components that satisfy the requests is employed. Each proxy server in our system stores some local information about the code components that are available in the shadow base, where code components of different resource usages and functionalities can be selected. The proxy system also applies a technique similar to active transcoding, where requests are modified before sending to the shadow base for selecting service codes of the suitable resource usages.

Different from other transcoding proxies, Mowser also considers the user preferences apart from the capabilities of the client devices. This is similar to our proxy system in which personalization is achieved by selecting service codes that satisfy the device user.

2.1.1.4 IBM Transcoding Proxy

IBM Transcoding Proxy [40] is also an http proxy that can transcode web contents for adapting to pervasive devices. It uses an *InfoPyramid* [18] as a data model to manage different *modalities* (e.g. text, audio, video) and fidelities (e.g. compressed image, summarized text) of the multimedia contents. Besides managing the multimedia contents, the InfoPyramid also manages the transcoding methods for generating the different versions of the contents. *Translation* and *summarization* are the two main transcoding methods that are managed by an InfoPyramid. Translation is a conversion of multimedia contents across the InfoPyramid, where a content is changed to a different fidelity (e.g. image $\rightarrow$ text). On the other hand, summarization is a conversion up the InfoPyramid, where a lower fidelity of a content is generated (e.g. an image of a lower resolution). Different transcoders can be used to translate and summarize web contents for adaptation, which includes compression, size reduction, color reduction, substitution, and removal of contents.

The modalities in content adaptation can be compared to the functionalities provided by the service codes, while the fidelities can be compared to the service codes of different resource consumptions. However, this direct mapping to functionality adaptation cannot be applied because a service is characterized by its functionality, which cannot be changed so as to satisfy the client. Translating to a different functionality implies providing a different service, which differs from the client's requirements. In the other dimension, summarizing a functionality requires semantic understanding of the service codes, and is impossible to be done automatically in the proxy servers.

2.1.1.5 Summary

To summarize, two main techniques are used in content adaptation: selection and transformation. For selection, servers maintain multimedia contents of different presentation formats (or modalities), each with different variants (or fidelities). The servers or intermediary proxies are responsible for selecting a content of a certain fidelity that best suits the clients. On the other hand, transformation does not require servers to store multiple versions of a multimedia content. Contents of a suitable format is generated on-the-fly according to the requirements of the clients. The advantages of this approach are that storage space in the servers can be saved, and contents generated on-the-fly can be made more suitable for the client compared to selecting one that is available from the server.

Despite of the advantages of transformation, this technique is not suitable for our proxy system. Service codes to be adapted are required to be executed in the resource-constrained devices for providing the desired services. Functionalities provided by these service codes cannot deviate from what are required by the clients. Unlike web contents, no information of a service code is trivial to be sacrificed while preserving the functionality, and transformation is, therefore, not suitable for functionality adaptation. Service codes that provide the same functionality, but with different dynamic resource usages are, therefore, needed for functionality adaptation. They are selected to be used by clients of different capabilities.

2.1.2 Code Adaptation

Code adaptation is quite rare in the field of pervasive computing, in which codes are adapted for execution in resource-constrained devices. Instead, most of the recent efforts are placed in content adaptation to adapt multimedia web contents to heterogeneous client devices. We have already seen some examples from the last subsection.

Although we can hardly find any systems for adapting codes to different devices, there have been a few effort [5, 16, 11] trying to modify program codes for other uses. They are studied for the feasibility of applying the techniques to code adaptation in pervasive computing.

2.1.2.1 Code Instrumentation

DynInst by the University of Maryland [5] and Binary Component Adaptation (BCA) by the University of California Santa Barbara [16] are attempts to modify program codes on-the-fly. DynInst provides a C++ API for dynamic instrumentation of codes into a running program. *Code snippets*

can be inserted into a running program for changing a code behavior. Developers do not have to re-compile, re-link or re-execute the programs to change the executables. This greatly simplifies the normal cycle of a program development. Similarly, BCA allows Java codes to be modified dynamically. Instead of modifying it in the source-code level, instrumentation is done in the byte-code level. Codes are adapted during program loading, by means of a *delta file*. The delta file contains information about the modifications to be made (e.g. addition of a method) in binary format. These modifications are written as an *adaptation specification* that is compiled to form the binary delta file, which is to be merged with the original program binaries just before they are loaded.

Code instrumentation techniques are mainly used by programmers to simplify their program development processes. Codes are added to modify the behaviors of the programs to meet their requirements, instead of reducing the executable sizes of the codes. This diversity of aims implies that code instrumentation is not suitable for adapting service codes for execution in a variety of client devices. Furthermore, code instrumentation usually results in the behavior being changed, which is not desired in our case. The necessity of human intervention is also another declining factor that makes code instrumentation unsuitable to be used for functionality adaptation.

2.1.2.2 LAURA Experiment

The LAURA Experiment by the NASA Ames Research Center [11] aims at re-coding programs in order to reduce the in-core memory usage in large programs. They carried out an experiment using a Fortran program called LAURA to run on NCSA's (The National Center for Supercomputing Applications) Cray Y-MP. The program was modified to reduce the use of the machine's main memory during execution. *Arrays* are used and stored in a high-speed DRAM-based auxiliary memory called Solid-state Storage Device (SSD). These arrays help reducing the executable size by retrieving and rewriting them in small sections, so that less main memory is used during program execution [11].

Its aim of reducing the executable size so that program codes can be executed with less memory is in-line with functionality adaptation. However, such a re-coding of program requires a thorough understanding of the program code in order to suggest a suitable data structure for reducing the executable size, and cannot be achieved in the proxy system without any human intervention.

2.1.2.3 Summary

There are rarely any researches that adapt codes to different devices for execution. In that case, efforts in code adaptation in other areas have been studied. Among them, the LAURA Experiment is the only one that shares a similar aim as our proxy system in adapting codes — reducing the executable sizes of the codes so that codes can be executed in the device. The only difference is that the experiment is only targeted to a particular machine instead of a wide variety of devices. From these studies, it can also be seen that most of the code adaptation techniques require understanding of the semantics at certain level, and is difficult to be achieved automatically.

2.2 Lookup Services

Services provided in a pervasive computing environment are quite dynamic due to constant updates of the services and their codes. The burden of locating services should be taken up by some intermediaries, such as agents or proxies. Users should only be responsible for describing the services that are required. The act of locating services are usually achieved by lookup services. Some lookup services [32, 42, 14] have been studied and compared with our proxy system in locating a service.

2.2.1 Jini Lookup Services

In a network-centric architecture like Sun Microsystems Jini [32], lookup services are usually used for locating suitable services. In Jini, clients first use a discovery protocol to discover a lookup service, and then send their requirements to the lookup service. The requirements are specified in a *service template*, which is a structure-like data model indicating the search criteria. Services that match the criteria specified in the service template are returned. The matching process is basically a *type-based match*, i.e. services of the same type as that indicated in the template is matched. To be more exact, the matching is based on three criteria. A service item is said to match a service template if [10]: (1) the service IDs of both the service item and the template are equal (or no service ID is specified in the template), and (2) the service item corresponds to an instance of every service types specified in the template, and (3) the attribute set of the service item contains at least one matching entry for each template that is supplied. Exact matching is used to match the attributes unless a wild-card is specified. Service items that satisfy the above criteria are returned to the clients. The clients can then choose a service among the returned items and download the corresponding service object. The service object contains method interfaces to be used for accessing the service.

Jini's lookup service is similar to the selecting of service codes for the desired functionality done in our proxy servers. They both locate services for the clients, according to their specified requirements. Jini uses a service template, while we use a service description (in the form of a query) for specifying the search criteria. For the matching process, the techniques are similar: different requirements are matched differently. In Jini, matching of the service type uses the concept of "instanceOf", and a similar approach is used for matching service codes that provide greater capabilities; while both use exact matching for matching of attributes. Despite the similarities, there is a difference. Jini only aim at finding services for the clients, but not selecting suitable services for them. It leaves the decision back to the users. On the other hand, our proxy system also takes into account the user preferences and other information about the execution context, so as to intelligently select a suitable service for a client.

In addition, Jini's lookup service also has its limitations. A range of values cannot be matched with a single service template, while it is possible for us to match a range of values by specifying them as attributes in the query. Furthermore, Jini's lookup service is intended for local area uses, and is difficult to scale to the Internet. Our proxy servers, being able to co-operate with each other, are more scalable.

2.2.2 Directory Services

The Domain Name System (DNS) [42] and the Lightweight Directory Access Protocol (LDAP) [14] are common directory services. Services are provided by computing nodes that are globally distributed. These computing nodes need to co-operate with each other in locating a service. They are arranged in a hierarchical tree-like structure, reflecting geographical, political or organizational boundaries; and contain information about the nodes to contact for different services. Each node can be identified by a name, or a set of attributes forming a *distinguished name* (DN). For the case of DNS, exact matching of the mnemonic name returns the corresponding IP address. In LDAP, *search filters* are provided for matching entries in the computing nodes. Comparisons like AND, OR, NOT, EQUAL, GREATER_THAN, LESS_THAN, and SUBSTRING are supported.

The Chord System [30] is a peer-to-peer lookup service based on DNS. It makes use of the Chord protocol to efficiently locate a service in the decentralized network of peers. With the use of an m-entry routing table in each computing node, and taking as input an m-bit identifier, the Chord protocol allows the lookup service to be done by sending $O(\log N)$ messages to other nodes.

DNS and LDAP are protocols for locating services that satisfy the requirements of the clients.

Systems have to be built on top in order to compare the intelligence with our proxy system. However, we can still compare the search filters used in LDAP with the matching filters used in our proxy system. They are both used for pattern matching of the services. However, by providing different kinds of filters, they support a better pattern matching than ours. In fact, ours is only a simple prototype at the current moment for demonstrating the feasibility of having a proxy system for adapting service codes in pervasive computing environments. We do not aim at implementing a full set of matching filters in our prototype. These filters can be added in the real system for better matching of the services. The Chord System aims at having an efficient lookup service while using DNS as its basis. Exact matching is used during the lookup process, while ours incorporates exact matching, range matching, and subset matching to identify service codes that satisfy the requirements of the clients.

2.2.3 Summary

Lookup services are required for locating network-centric services for the clients. Jini has a lookup service in which our proxy system shares the most similarities. Requirements are specified as properties to be used as searching criteria, and matching of these properties is needed in locating the services. DNS and LDAP define the protocols that require systems on top to provide more intelligence. Only basic operations for matching services are defined. All these lookup services are only responsible for locating services for the client, and leaves the selection decision to the users. Our proxy system, besides locating services, also selects a suitable service to the client, according to the user preferences and other information about the execution context. This kind of personalization has also been used in content adaptation systems, such as iMobile [24] and IBM's transcoding proxy [40], that return customized contents to the client. User preferences are usually used for achieving personalization.

2.3 Other Features for Pervasive Computing

Pervasive computing is not a totally new topic. Many approaches have been suggested for pervasive computing, including Web Services. There are also issues related to pervasive computing, such as adaptation of web contents to heterogeneous client devices, that are extensively discussed in literature. Besides these adaptation issues and the service lookup issue that we have studied, we are also looking for other desirable features to support pervasive computing.

2.3.1 User Mobility

To our understanding, user mobility is the ability of a user to move and compute, while being treated as the same user independent of the physical location. Any kinds of support, such as maintaining the same user environment, that helps to treat the same user in different locations as the same person without affecting his/her computing experience are considered as user mobility support. User mobility can be supported in the proxies or the middleware systems. Different systems have been designed to support user mobility with different approaches [28, 31].

2.3.1.1 Aura

Carnegie Mellon University's Aura [28] is an architectural framework for supporting user mobility in ubiquitous environments. It acts as a *personal proxy* for the mobile user it represents. When the user leaves an environment, Aura suspends the on-going task, requests all the states of the services that are on-going to be checkpointed, and deallocates these services and stores necessary files to the Aura file space. As the user enters a new environment, Aura then sets up the environment accordingly and *resumes the suspended task*. In the new environment, different devices might be used for continuing the suspended tasks, and dynamic reconfiguration needs to be supported in Aura for accomodating to the dynamically-changing resources. *Connectors* may also be used between different component types in Aura for interconnection.

Similar to Aura, proxy servers in our proxy system also act as personal proxies for the users. They maintain personal information such as user preferences, user past histories, and personal caches for caching service components that the users require. Instead of using the information for providing a uniform environment for the user at different locations, it is used for selecting personalized services for the user. For the case of continuing the on-going services in another location, the effect of shielding the users from the heterogeneity and dynamic variability of device capabilities and resources can also be provided in our proxy servers by co-operating with a Mobile Agent System.

2.3.1.2 WebMC

Web-based Mobile Computing (WebMC) [31] by the University of British Columbia is a web-based middleware solution for supporting user mobility. It supports user mobility by retaining the user's *Personal Computing Environment (PCE)* no matter where the user is and what device the user is

using. A user's PCE stores the user's personal document, favorite applications or services, personal preferences, etc. When a user moves to a new computing environment, the WebMC in the new environment contacts the home server for the user's PCE, and sets up the same environment for the user, according to the information in the PCE. In order to hide the user from the heterogeneity of the OSes that different devices are using, WebMC provides a web browser interface for users to access the applications or services. Users can simply click on the corresponding link for the service, so that WebMC can download the relevant files and finds a suitable application locally for providing the service. By doing so, WebMC tries to integrate local resources in supporting user mobility.

Different from Aura, WebMC supports user mobility by imposing uniformity on the environment, through the use of PCE. Users computing in different environment shares the same personal environment in accessing services and applications. However, instead of providing the same environment for accessing services, our proxy system focuses on setting up a better implicit environment for the clients. This implicit environment refers to the environment in the proxy server that affects the clients' executions, such as the personal user cache. By setting up this implicit environment for the clients, user-perceived latency experienced by the movement should be better reduced, providing a support of user mobility in that sense.

2.3.1.3 Summary

In a pervasive computing environment, there is no fixed association between the device user and a mobile device. Users can move without carrying any devices and compute with any device that is available in the new environment. This change of environment is usually experienced by the user, due to the variations and differences in resources, applications and settings. Aura and WebMC are two attempts to alleviate this problem by providing the same computing experience for the user independent of the movement. Different approaches have been used to tackle the problem. Aura provides an architectural framework that can be applied to an intermediary (e.g. an http proxy), and focuses on the continuation of on-going tasks; while WebMC provides a middleware solution that runs on the client devices, and focuses on providing the same personal environment for the user.

With the help of the Mobile Agent System, continuation of on-going services can be supported in our proxy system. This shares the same idea as Aura in its ability to resume suspended services in a new environment. However, our proxy system does not aim at providing dynamic reconfiguration of the service components for handling the dynamic variations of the capabilities and resources in the environment. Instead, service components are on-demand downloaded when they are needed,

and the proxy system only selects service components according to the client's execution context upon request.

In setting up the environment for the user in a new location, WebMC makes use of the concept of a personal computing environment. In our case, this personal environment is the personal user cache in the nearest proxy server to the client. Instead of using the same personal cache in all proxy servers, each proxy server maintains a local personal cache for the user. This stems from our belief that different services are required in different execution contexts. There is no need to maintain the same cached contents in every proxy servers. The personal caches should, instead, be pre-loaded with service components that are expected to be used in the near future.

2.3.2 Recommender

With pervasive computing, numerous services are available everywhere for the clients to access. These services are added or removed as time goes by, for software updates or other maintenance. In accessing updated services, a selection decision is needed every time when a service is required. This becomes very tedious if the decision has to be made by the user everytime. Using intelligent entities as recommenders solve this problem. Information for making the decision, such as user preferences and execution context, can be given to the recommender. The recommender can then selects (or recommends) a best-fitting service on behalf of the clients.

2.3.2.1 W^3IQ

W^3IQ [15] is a system for supporting mobile web access. Its objectives are to manage disconnections and bandwidth-saving, the main problems in mobile computing environments, by acting as a recommender to provide the right information for the user. In *disconnected browsing* (information retrieval from WWW in mobile environment), users specify the requests as queries and disconnects. The W^3IQ proxies process these requests through *information filtering* and *collaborative information retrieval*. Results of the request are available when a user connects again. These resulting URLs that satisfy the queries can be ranked by the users and stored in the user's profile for answering future queries. Therefore, the W^3IQ system utilizes and reuses information that already exists in the system in providing the right information for the users.

In recommending information or services to the client, both W^3IQ and our proxy system require information from the users (stored in user profile). However, by requiring users to explicitly rank the URLs for each request, W^3IQ requires more interaction with the users, which makes it

more tedious. On the other hand, our proxy system uses information of the user preferences, which can be the same for general requests, that has been given to the proxy when the user first registers. Besides using user preferences for the recommendation, our proxy system also uses other information, such as the execution context and proxy preferences. Furthermore, the proxy servers can also co-operate with each other, like the collaboration of the W^3IQ proxies, when a proxy server cannot handle the request alone.

2.3.2.2 SiteSeer

SiteSeer [25] is a commonly-sited recommender system in recommending web pages to users. It makes use of the *bookmarks* and the organization of bookmarks (using folders) that have been maintained by the users to predict and recommend web pages personalized for the users. These bookmarks are means of revealing user interests in the web pages, and are suitable to be used for prediction. Besides that, SiteSeer also predicts relevant web pages information from their neighbors, and treats them as implicit recommendations for the users. To provide this implicit recommendation, it measures the *degree of overlapping* of the web pages (within a category) with other users to identify the similarities among individuals. Web pages within a category of a neighbor, that has the most web pages overlapped with the user, are used as implicit recommendations for the user.

Unlike SiteSeer, our proxy system does not compare the similarities of the service components among individuals. Different users, in different execution contexts, are believed to have different execution behaviors. These differences in execution behaviors result in different services being requested, making it unwise and useless to compare service components that have been used by different users.

2.3.2.3 Summary

W^3IQ and SiteSeer are recommendation systems aimed at recommending information to the user, so as to achieve personalization. W^3IQ is developed for the mobile computing environment, and focuses on using collaborative information retrieval in retrieving the correct information for supporting disconnected browsing. SiteSeer, on the other hand, is not developed for mobile computing. It is simply a personalization technique for traditional web browsing. It makes use of personal bookmarks and the overlapping of web pages with other individuals to identify similarities for implicit recommendations. Despite of the different approaches in making the recommendations, the two recommendation systems and our proxy system all make use of user preferences (either implicit or

explicit) to make decisions for the users.

Chapter 3

Overview of the Sparkle Project

Before going into the details of the proxy system, this chapter gives you an overview of the Sparkle Project, which is our attempt to build an infrastructure that is suitable for pervasive computing environments. This overview shows the computing environment that our proxy system assumes, and acts as a preliminary for better understanding of the details in the coming chapters. Besides that, some terminologies that are critical to the understanding of the mechanisms in later chapters are also defined in this chapter.

3.1 Architectural Overview

Similar to other projects like Microsoft .NET [43], Sun Microsystems Open Net Environment [45], and the University of Washington's One.World [49], the Sparkle Project aims to build an infrastructure that is suitable for pervasive computing environments. The infrastructure is based on the existing Internet infrastructure, with *adaptability*, *mobility support* and *peer-to-peer co-operation* as its main features. These three supports are important for infrastructures in pervasive computing environments. Adaptability addresses the problem of computing with heterogeneous devices in different execution contexts. Mobility support addresses the problem of continuing the current session in another device or another location. Peer-to-peer co-operation avoids single point of failure by allowing services to be provided by nearby peers in the case of network disconnection.

In order to support pervasive computing, services in our infrastructure are composed of mobile code components, called *facets*, which are on-demand downloaded to the client devices, executed and then discarded. The constituent code components of a service are not fixed at compile-time, but are dynamically bound to form a service. Facets are mainly downloaded from proxies, but can

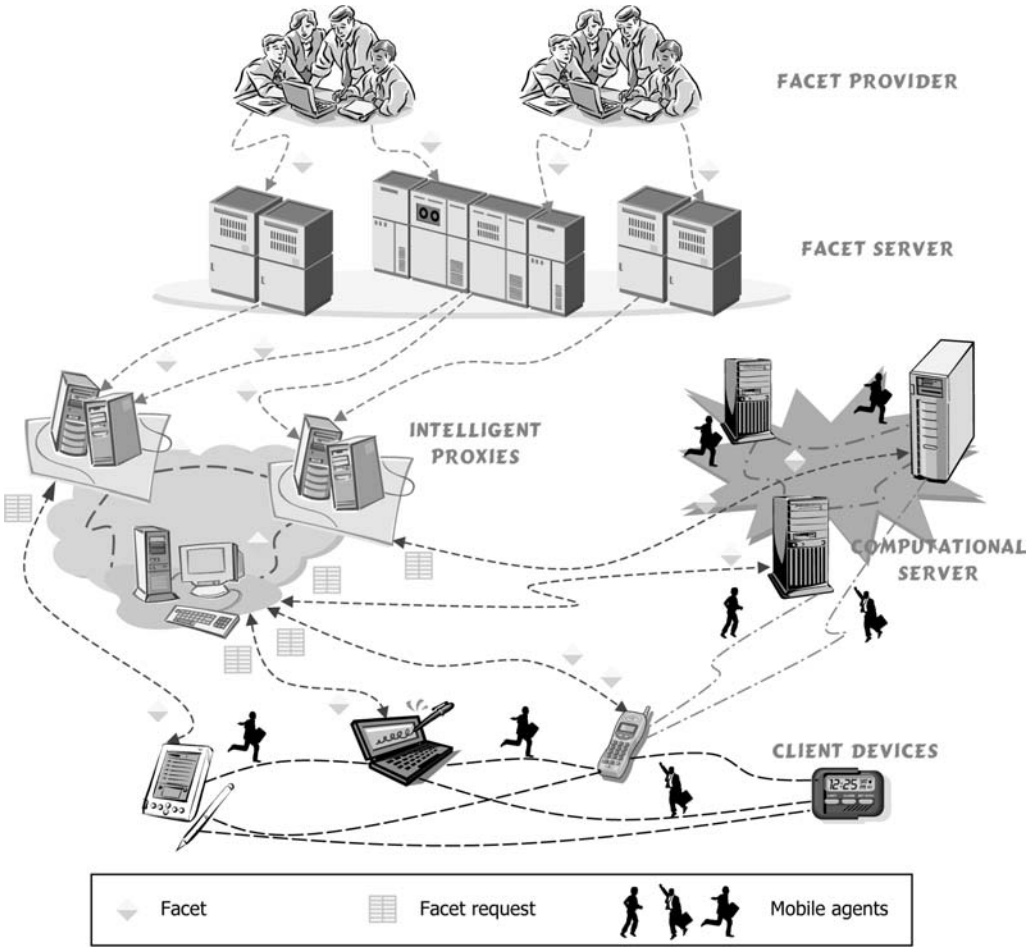


Figure 3-1: An architectural overview of the Sparkle system

also be downloaded from nearby peers in case of disconnection. Clients, therefore, can work as individuals and get the facets from proxies; or work as a peer group and get the facets from their peers. Figure 3-1 shows the architectural overview of the Sparkle system.

Similar to other 3-tier infrastructures, facet servers, clients, and the intermediary proxy system are the three main components of Sparkle. Each of them plays a different role in the infrastructure. Here, we only give a brief introduction of each of them. For details, please refer to [2].

3.1.1 Facet Servers

Facet servers are the places for storing facets. They are similar to the existing web servers in that both are used as main storage servers that contain the up-to-date originals. They are used by the proxy servers for retrieving updated information. There is no restriction to the number of copies of a facet to be placed on these servers. A facet can be placed on more than one facet server, meaning

that facets are not unique among the facet servers.

Facets can be added to or removed from the facet servers by facet providers. These updates are usually quite frequent in terms of software maintenance. In order to keep track of the updates made to the facets being stored, each facet server needs to keep a log of updates of the facets they store locally.

3.1.2 Clients

Clients in our infrastructure can be any computing devices, ranging from large stationary computers to small mobile devices such as Personal Digital Assistants (PDAs) and smart phones, independent of their form factors and computing powers. Each of these computing devices has the capability of on-demand downloading, executing, and discarding of facets after use, thereby allowing services of any sophistication to be executed on the client device.

Whenever a service is required, request is sent to a nearby proxy for a suitable facet. The request consists of a description of the required service, information about the resources in the client device that can be used for executing the required service, as well as some user information. A facet satisfying the request is then returned to be executed. During execution, other sub-services might be required to help providing the service. Requests are then sent to the proxies when these sub-services are needed at run-time.

Apart from this, each client device is also installed with a mobile agent system to help supporting mobility and peer-to-peer co-operation. Mobility can be supported by migrating on-going tasks to another device (with the use of *containers*) and continue execution. For peer-to-peer co-operation, mobile agents are sent to nearby peers for facets in case of network disconnection. Details about the lightweight mobile agent system can be found in [7].

3.1.3 The Proxy System

The proxy system is a main component between the clients and the facet servers. Facets are cached in the proxies for fast retrieval and to reduce the workload of the facet servers. Client requests for facets are therefore sent to the proxy system instead of directly to the facet servers. Apart from being a caching device, the proxy system also acts as a *recommender*. It makes decisions on behalf of the clients and returns a suitable facet for each request according to the run-time execution contexts of the clients.

Being able to choose a suitable facet for the client makes the proxy system intelligent. It is this intelligence that makes the facets able to be executed and provide the required services in the client devices, thus making the proxy system the key-enabler in the Sparkle Project. Without this ability, services cannot be accessed from any device, and pervasive computing cannot be supported.

In order to allow more flexibility for the proxy system to choose among the facets, requests are specified in terms of queries instead of exact locations of facets. Furthermore, the proxy system maintains personal proxy caches for the users, so that facets can be better adapted to the device user. With all these supports, adaptability can be better provided by the proxy system. On top of these, the proxy system also supports user mobility by co-operating with the lightweight mobile agent systems in the client devices. With this support, the same user in a different location is possible to be treated equally independent of the location, without affecting the computing experience. On-going services are, therefore, possible to be continued in another device with suitable facets adapted. The proxy system also prepares the personal proxy cache that is to be used with suitable facets in supporting user mobility.

3.2 Comparison with Web Services

Allowing mobile code components to be downloaded and adapted to the client devices is not the only way to support pervasive computing. The recent emergence of Web Services [37, 48, 13], which makes use of dedicated web servers for providing services to clients, is another. Both approaches have their own debits and merits. This section compares the Sparkle Project and the Web Services.

For Web Services, services are provided by dedicated servers that execute service codes on behalf of clients. Data required to provide the services is passed to these servers for processing, and the results are returned to the clients. Since only the resulting contents are returned, adaptation entities (e.g. intermediary proxies) only concern about the presentation formats and the static resources of the resulting contents, but not the dynamic resources that the execution consumes. These resulting contents can be easily adapted to different client devices using existing content adaptation techniques, and Web Services is, therefore, suitable to be used for pervasive computing. In Sparkle, mobile code components are used, so that codes can be executed on the client devices and, therefore, do not require data to be passed to other network entities for processing. This is good for protecting users' privacies. However, this approach requires mobile code components to be adapted to the resource-constrained devices, which is much more difficult than adapting the resulting contents. It

depends on the dynamic resources the service codes consume, which can only be determined at run-time.

Another difference between the two approaches is in the use of proxies. Web Services have dedicated web servers for providing services and, therefore, is not necessary to have proxies for caching data and redirecting requests. On the other hand, proxies in Sparkle are needed for caching mobile code components to improve the access latency and reduce the workload of the facet servers.

In order for others to know what services are available, Web Services usually have a registry for registering and locating these services. The registry is just like a "lookup" service. It searches the database of registered services and returns a list of services that satisfies the query, relying on the clients to choose a suitable service themselves. So, for a client to access a service, it usually needs to send at least two requests: one to the registry, and another to the actual service provider. Sparkle simplifies this two-step process by introducing intelligence to the proxies and allows them to make decisions on behalf of the clients. Besides taking the burden of selection away from the users, this approach has the additional advantage that facets can be adapted to the surrounding context (e.g. by identifying the similarities of the services needed by nearby users in the same group), which makes them more suitable for the clients. This kind of context-aware adaptation is difficult for the case of Web Services.

3.3 Terminologies and Definitions

Some terminologies are used extensively throughout the whole thesis that require clear definitions for better understanding of the mechanisms. In this section, we will introduce some of these terminologies.

3.3.1 Terminologies Related to Facets

Mobile code components that constitute services in our system are called facets. They are mobile, and can be placed on any well-known facet servers to be downloaded. With the help of proxies, facets are downloaded to the client devices for execution. Each facet, when executed, performs a specific functionality for the clients.

A facet does not need to perform the whole functionality by its own. It might need other facets to help achieving its functionality. During its call for other facets, its specified functionality has not been completed. The facet is still being used and is said to be ***active***. After being used, the facet

becomes inactive and can be discarded. There may be several facets having the same functionality, but with different implementations and run-time behaviors. Facets are said to be ***compatible*** if they carry the same functionality. Proxies can then choose among these compatible facets for adapting to the clients.

In order to help determining whether a facet is suitable to be returned or not, each facet needs to have some descriptive information in addition to the code for providing the service. Therefore, a facet is consist of two parts:

- **Code segment.** This is the part where the executable code lies, and contains only one publicly callable method to be called upon by others. This code, when executed, should perform a pre-defined specific functionality.
- **Shadow.** This is the meta-data for describing the facet. Without this, a facet is just a piece of executable code and cannot be distinguished. It specifies the properties of a facet, such as the vendor, version, the functionality it performs, the resources it needs in providing the functionality, and the functionalities that the facet requires for execution. All these information helps the proxies in choosing a facet for the clients.

3.3.2 Functionality and Service

Functionality is the execution behavior that a facet performs when it is being executed. It can be seen as a contract between the facet provider and the clients, which includes:

- a *description* of the execution behavior that the facet is required to perform,
- the *input* to be taken in for processing,
- the *output* to be produced,
- the *pre-condition* that the facet assumes for achieving the specified behavior,
- and the *post-condition* that the facet assumes after completing the execution.

The resulting behavior of a facet can only be uniquely identified by specifying all the above information. For example, a facet with only a description of “rendering an image”, taking an image as input, and produce an output image, cannot clearly specify the resulting behavior that the facet achieves. A facet that renders an image might only be capable of rendering an image specified by a bitmap; while another facet might only be capable of rendering a jpeg image. So, the pre-condition

Description: PostScript to PDF converter **Input:** A string **Output:** A string **Pre-condition:** The input string contains characters in PostScript format, and cannot be longer than 1024 characters **Post-condition:** The output string contains characters in PDF format

Figure 3-2: An example showing the functionality of a facet.

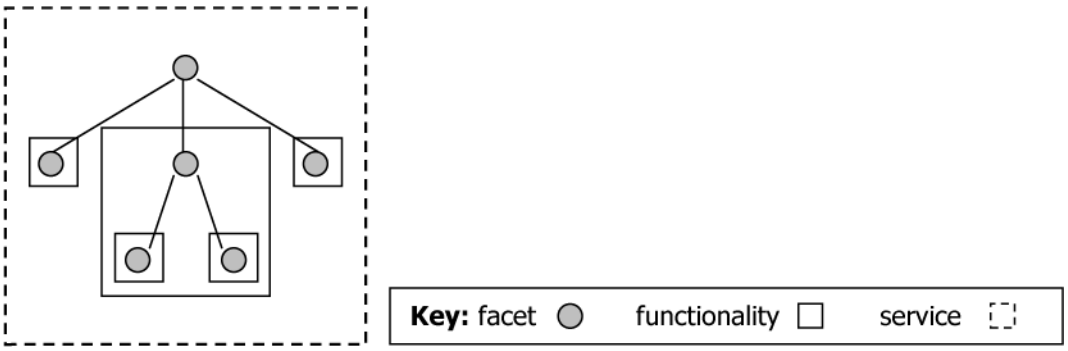


Figure 3-3: Demonstrating a service and a functionality.

and post-condition that the facet assumes are also needed to be specified in order to clearly state its resulting behavior. Figure 3-2 gives an example of the functionality of a facet.

Functionalities, being specified, can help the proxies in selecting facets for the clients. Users can also use these information to describe the functionalities they need instead of specifying a specific facet to be used for achieving the functionality. These functionalities, from the users' point of view, are called ***services***. E-mail, editing, file compression, and image viewer are examples of services. Figure 3-3 shows the concepts of a service and a functionality.

3.3.3 Terminologies Related to Facet Dependency

As mentioned before, a facet can call upon other facets to help achieving its functionality. In order to be more flexible, facet providers do not specify the facets it requires. Instead, they specify the functionalities that the facet needs, to allow facets to be dynamically bound. These functionalities required by the facet are called its ***facet dependencies*** (or simply, dependencies). Facet dependencies, therefore, represent a *local point of view* of the facet. The dependencies that a facet depends on can be represented as a ***facet dependency tree***, as shown in Figure 3-4(a). Facet dependency trees are only one level, as a facet only knows the dependencies it requires.

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

For better illustration, let us take a look at an example. Assuming a facet for viewing an image, say F_{view_1}, requires other functionalities like reading an image from a stream and decoding an image. At run-time, F_{read_2} is used to read the image from a stream. After reading the image, the image needs to be decoded and F_{decode_3} is used for decoding the image. However, F_{decode_3} does not perform the functionality all on its own. It requires other algorithmic functions (provided by other facets) to help decoding the image. For simplicity, we assume that $F_{algorithm_1}$ is used for achieving this purpose. In this example, reading an image from a stream and decoding an image are the only dependencies of F_{view_1}; while the algorithmic functionalities for decoding an image are the dependencies of F_{decode_3}. Figure 3-4(b) shows the execution tree of F_{view_1}.

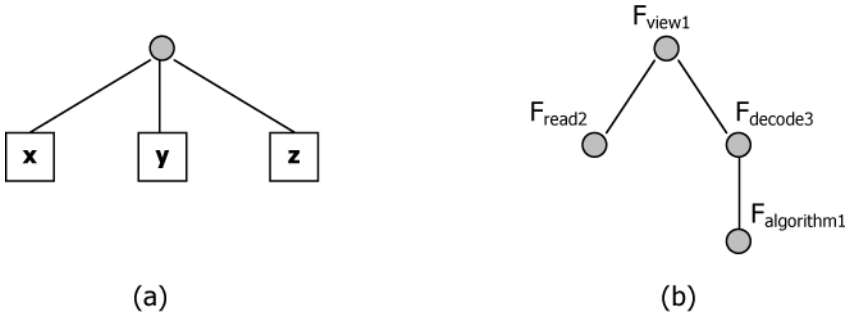
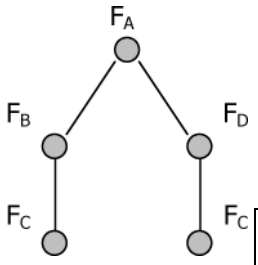


Figure 3-4: (a) A facet dependency tree and, (b) a facet execution tree.

From the figure, F_{view_1} (the root of the facet execution tree) is called the ***root facet***, and the depth of the tree is called its ***calling depth***. All the other facets of the execution tree are used to help the root facet to achieve its functionality.

From a facet execution tree, we can also identify the ***facet calling sequence***. A calling sequence is a sequence of facets being used to perform a functionality at run-time. When only a single service is concerned, it is the sequence of facets used to achieve its functionality, which can be considered



Facet calling sequence: $F_A \rightarrow F_B \rightarrow F_C \rightarrow F_D \rightarrow F_C$
Facet request sequence: $F_A \rightarrow F_B \rightarrow F_C \rightarrow F_D$

Figure 3-5: A facet calling sequence indicated in a facet execution tree.

as a *traversal* of the facet execution tree in some order. For simplicity, the facet execution tree is usually drawn such that the calling sequence can be found by an *in-order* traversal of the tree. Another sequence, called a ***facet request sequence***, is the sequence of facets being requested to perform a functionality at run-time, and is generally a *subsequence* of the calling sequence. It can be, or can not be, the same as the calling sequence, depending on whether facets have been cached. If facets have not been cached in the client, the request sequence is the same as the calling sequence. Otherwise, the request sequence becomes a proper subsequence. Figure 3-5 shows a calling sequence and a possible request sequence.

In the figure, we have assumed facets have been cached. The request sequence only consists of four facets, due to the fact that facet F_C has been requested and cached, and need not be requested again.

Chapter 4

Functionality Adaptation

After having a brief idea of our proxy system and its working environment, this chapter highlights the focus of the proxy system — functionality adaptation. We first describe the need of functionality adaptation in pervasive computing environments. After that, we define the scope of functionality adaptation that our proxy system concerns, and discuss the conditions affecting the adaptation. Lastly, techniques we use for achieving functionality adaptation would be presented.

4.1 The Need of Functionality Adaptation

The future trend of pervasive computing, in which computing can be done anytime, anywhere, and on any device, differs greatly from the traditional way of computing. Users are no longer required to stick to their PCs at home or in office for computing, but can also use their mobile devices, or even devices that they come across. For example, they can use the computing devices in any coffee shop they pass by, or simply borrow a device from a neighbor when their devices do not have enough computing resources. This means that they can use any devices that come to their hands, but not necessary their own personal devices.

In addition, due to the inherent nature of pervasive computing, computing devices ranges from stationary PCs to small mobile devices. Each of these devices has different hardware and software capabilities. Technologies to be used for pervasive computing, therefore, have to take into account the heterogeneity of the computing devices.

Besides adapting to the client devices, our proxy system also needs to adapt to the device user. Users have different user preferences, which depends on the services required, physical location, resources nearby, etc. Therefore, different users using computing devices of the same capability

should also be treated differently, and to be provided with a personalized service (or functionality) that best suits them.

Functionality adaptation, therefore, needs to be provided in order to enable pervasive computing; with the functionality (or service), resulting from the execution of the service codes, adapted to the clients.

4.2 Scope of Functionality Adaptation

Recently, works have been focused on designing platforms and architectures for pervasive computing [12, 34]. Although each of them has its own features, they tend to follow the so-called "Web Services" approach. Clients are not expected to perform much computation, and are usually resource-limited. Instead, computations are performed in resource-rich servers, and made themselves known to the clients as "services".

Services that are monolithic in nature are not very usable, as every service provider has to write the entire service by their own. In order to make Web Services usable, services are usually componenized. Each component performs a specific task, and several of these components are used to form a service. In order to allow more flexibility, services can be dynamically composed, in which the constituent components are the results from the capabilities-based lookup at run-time. This dynamic nature allows different kinds of services to be composed to suit the clients.

Although Web Services is a common approach for pervasive computing, it suffers from a major problem concerning privacy in that data needs to be given to the servers for computation. This makes downloading code components for execution another alternative. Instead of handing data over to other places for computation, code components are downloaded to the client devices for execution. Functionality adaptation we discuss is, therefore, base on the infrastructure of the Sparkle Project described in Chapter 3, which assumes that mobile code components are executed in the client devices, and then discarded after use. Functionality adaptation should be able to extend to other models that allow code components to be downloaded to the client devices for execution.

In our model, services are make up of facets, and can be dynamically composed. The model is so dynamic in the sense that facets are dynamically bound to their functionalities at run-time. A facet only needs to specify its required functionalities, but not the facets. Therefore, the facets that are used to perform a functionality (or actually, the facet execution tree) are only known at run-time. Despite that, clients still expect the functionality (provided by a sequence of facets) to be completed

in the client device, once it has started execution. Therefore, the proxy system needs to adapt the services, or in general, the functionalities, to the client devices. The facets returned should be able to execute in the client devices, and provide the desired functionality. The client devices should not run out of resources in the middle of an execution.

Besides adapting to the client devices, the adaptation should also take care of the device users. With the enormous and frequent updates of the facets, it is difficult for users to keep track of the available facets. Therefore, in our model, the proxy system acts as a broker, as well as a recommender, to help clients select a facet that best suits them.

To be more concrete, functionality adaptation in our model is to adapt facets to clients, such that:

- the desired functionalities can be achieved in the client device, and
- the facets adapted should best suit the device user.

4.3 Context and Context-Awareness

In order to adapt the functionality to the device user, the proxy system needs to be able to select a suitable facet. This selection requires knowing additional information so as to make the proxy system aware of the situation a client is in, and thus helps making the decision. This additional information includes the amount of resources available for executing the desired functionality, the location of the client, users nearby, and some user information such as the user preferences and his past usage pattern. The proxy system can then make use of these information to select a suitable facet for the clients.

In traditional computing environments, the above-mentioned information is almost static, and can be used by the proxy system to determine the future. However, in pervasive computing environments, these information changes all the time. It is not possible for the proxy system to use previously available information to determine the future. Clients need to send up-to-date information so that the proxy system can be aware of the client's run-time situation and make suitable decisions for them. Therefore, the proxy system needs information about the run-time computing environments of the clients in order to make suitable selection decisions.

4.3.1 Definition of Context and Context-Awareness

The above information that describes a client's situation is known as the client's *context*. According to Dey and Abowd [8], context is "any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object". A client's available resources, location, nearby users, and the device user's information can be used to characterize the situation of the computing entity, and therefore regarded as context by this definition. Since the context we need are only those that affects the client's execution, we refer to them as the client's ***execution context***:

> *Execution context is any information that can be used to characterize the execution of the client.*

Besides context, Dey and Abowd also define *context-awareness* to be "the use of context to provide task-relevant information and/or services to a user". This definition suits to describe our proxy system, which aims at adapting the functionalities provided by the facets to the clients. In order to make suitable decisions on behalf of the clients, the proxy system needs to be aware of the clients' execution contexts. Knowing these contexts helps to provide suitable (and personalized) services to the users, and the proxy system is, therefore, said to be context-aware.

4.3.2 Relation to Functionality Adaptation

As mentioned before, clients' execution contexts affect the proxy system's decision in selecting suitable facets for the clients. The decision is affected because of the change in the execution behaviors of the clients in different contexts. In fact, execution behaviors of the clients are usually affected by their execution contexts, and is especially true when users are involved. This execution context can be the location, or the behavior of the nearby users. Location can affect the activities of the users, and therefore the services (or functionalities) being accessed. Users in different environments usually have different user activities. This can be seen from their usage patterns in different environments. For example, office-users normally require services related to their work, such as editing, drawing, e-mail, ware-housing, etc. Home-users, on the other hand, require services for leisure, such as watching movies, playing songs, reading comics and newspapers, playing on-line games, e-mail, etc. Although some of services might be needed in both environments, they are, in general, two different sets of services.

Besides location, the behavior of nearby users can also affect a client's execution behavior. For example, in a work-place, users might be working as a group, such as a graphics group. In such a case, they are likely to be using the same computing services (e.g. drawing). This is only true when the users are working together. If they are working as an individual, such as home-users, their execution behaviors are not affected by their neighbors.

Execution contexts do not only involve location and the users nearby, they also involve the capabilities of the client devices such as the resources available for an execution. The execution behavior of a computing device depends on its capability. A device with more resources (e.g. memory) should be able to execute service codes that consume more resources (which can be a sign of having better service quality), when compared with another of less resources. Therefore, context information is needed in order to select suitable facets for the best use of the computing device.

As can be seen, a client's execution behavior is greatly affected by its execution context. Knowing a client's execution behavior helps making suitable decisions. This is even more important in pervasive computing environments, as the execution context changes more frequently between requests. Being able to know the run-time execution contexts of the clients and respond according to them is needed in pervasive computing. Our proxy system, being designed for functionality adaptation in pervasive computing environments, is, therefore, context-aware. Functionality adaptation can thus be seen as a kind of context-aware adaptation, adapting service codes (facets) to clients according to their execution contexts.

4.3.3 Classification of Contexts

In general, contexts can be classified into different categories. There are no general rules for classification, and different people can have different ways of classification. For example, Schillit [26] classifies context into three categories: computing context such as network connectivity, communication costs, and nearby resources such as printers; user context such as user's profile, location, people nearby, and current social situation; and physical context such as lighting, noise levels, and temperature. Later on, Chen and Kotz [6] thought that time is also important but cannot be put into any of the above categories. They then base on Schillit's classification and add a fourth category: the time context.

This classification is not suitable to be used in functionality adaptation. In order to make it suitable, we classify it differently. We only focus on execution contexts, and classify them into two categories: *device-dependent* and *device-independent* contexts. Device-dependent contexts are

more easy to predict within a time period, when requests for facets that constitute a service are likely to come from the same device. On the other hand, device-independent contexts are more difficult to predict, as they may depend on user behaviors.

4.3.3.1 Device-dependent Context

Contexts that are device-dependent are related to the device capabilities. Examples of contexts in this category include the amount of resources available for execution, the processing power, the size of the screen, and its color depth. These limit the resources that can be consumed during execution, and the resulting contents that can be displayed. All these affect the service codes that can be downloaded and executed in the client devices.

In this category, the main concern is whether the returned code can be executed and completed in the client device. Others, such as whether the functionality is suitable or not, are not its concern. That means, it is only concerning the adaptability to the client device, which is similar to some existing adaptations, such as content adaptation [40, 23]. Functionality adaptation in this category, therefore, can be considered as having different versions of facets that provide the same functionality, each with a different implementation and run-time behavior, thereby using different amount of resources. Adaptation can then be achieved by selecting one of these variants that satisfies the client device's capability so that it can be executed and completed in the client device.

4.3.3.2 Device-independent Context

All other execution contexts that are not related to the client device capabilities are device-independent contexts. Examples include location, time, people and resources nearby, and user information such as user preferences and the user's past usage pattern. They do not affect the feasibility of the client device to complete the execution of the service codes. Instead, they change the execution behaviors of the clients, and thus the functionalities required. This affects the decision of selecting service codes that provide a suitable functionality for the clients in their execution contexts.

The main concern in this category is the suitability of the services. We need to be able to know the kind of services (and the service codes) that suits the execution context. Functionality adaptation in this category is, therefore, to select facets that provide a suitable functionality for these device-independent contexts. This requires the proxy system to have some intelligence in predicting the functionalities that the clients need.

4.4 Adaptation Issues

With all the necessary information, such as the functionality required, the amount of resources available for execution, and other execution contexts, the proxy system is able to adapt facets to the clients. But before looking at how functionality adaptation is achieved, a few issues need to be discussed because they affect the decision of which technique(s) is/are going to be used for functionality adaptation.

4.4.1 Code Adaptation

In the traditional Web model, web servers store data to be shared. Clients requiring the data have to download them to their devices. Therefore, whether the data can be presented properly in the client devices is their main concern. Even in the case of Web Services, codes are not their main concern. All the executions are performed on dedicated servers to enable computing in heterogeneous devices. This frees them from adapting codes to the client devices. They are only concerned with adapting the resulting contents, which are a kind of data, to the client devices. This data is usually a static one, and the resources it consumes can be statically determined and used for adaptation. In the case of dynamic data, the client device is usually assumed to have enough resources for the resource variations.

This is different from what we assume in our case. In our model, codes can be downloaded to the client devices for execution. Therefore, what we concerned in the adaptation is whether the codes can be executed and completed in the resource-constrained devices. The resources that a code consumes cannot be statically determined, as it depends on the control structures used in the code and, in most cases, the size of input data. The amount of resources a code consumes, therefore, can only be determined at run-time when the input is known. This dynamic behavior of the codes makes functionality adaptation, which is also a kind of code adaptation, non-trivial.

4.4.2 Adaptation Techniques

There are mainly two adaptation techniques adopted in content adaptation, namely *transformation* and *selection*. Transformation is to undergo some modification such that the modified contents are suitable for the client devices. Selection, on the other hand, do not involve any modification. Different variants of a content are prepared beforehand, and a suitable one is selected when requested. However, not both of the techniques can be used in functionality adaptation.

4.4.2.1 Transformation

Transformation is a common technique used in content adaptation. It is also known as transcoding [40, 46]. Originally, the data might be too large to fit into some small mobile devices, such as the multimedia files (e.g. the movies) and the web pages that are originally generated for the PCs. In order to allow the small mobile devices (with limited resources) to access these data that is originally quite large, data is transcoded before returning to the client. The transcoding is to remove syntactic information such that the data required can be presented properly in the resource-constrained devices, and at the same time, maintaining the same semantic information as the original content.

During transcoding, some information is intentionally removed (i.e. lossy) so as to reduce the data size and, thus, the resources it consumes. This information only affects the syntactic structure of the contents, but not its semantic meaning. For example, an image may be transcoded to one of a lower resolution; or a brief description of a video file is displayed instead of playing the video in a resource-constrained device.

Although transformation is a good approach in content adaptation, this is not the case for code adaptation. Codes are not for display, but for execution in the client devices. Any loss of the code structure during transcoding might result in an incomplete code that is unable to be executed. Even in a lossless transformation, it is difficult to guarantee the transformed code can achieve the same functionality as the original code. In order to determine whether the two are equivalent during execution, semantic analysis of the codes is required, which implies human intervention is needed. This makes the adaptation difficult to be performed automatically on-the-fly. The situation is even worse if the codes are in executable format, where source codes are not available for semantic analysis.

4.4.2.2 Selection

Using transformation requires some intelligence so as to remove some information that is not critical to the semantics of the contents. In cases where transformation cannot be used or not much intelligence is needed, selection is a possible approach [33].

For the case of selection, an application or a service usually appears as different component configurations, each suitable for a different range of devices. One of the available configurations is then chosen to be used in accessing the service from the client device. This approach has the advantage that selection is easy compared to transformation, as it only requires a comparison with the device's

capability. However, it also has a disadvantage that selection is base on the configurations that are available. These configurations cannot be tailor-made for each and every computing devices, and the one to be chosen is what appears to best suit the client among all the available configurations.

Despite of the disadvantage, selection is a possible approach for functionality adaptation. Functionalities are provided by executing the service codes. There can be different facets, with different implementations and run-time behaviors, achieving the same functionality. Each of these facets consumes different amount of resources during execution, and is suitable for only a specific range of target devices. Facets are selected at run-time according to the run-time execution contexts. Among the facets that are available to provide the desired functionality, the one that appears to best-suit the client is selected.

4.5 A Two-phase Adaptation

As discussed in the previous section, selecting a suitable configuration for the required service is a possible approach for functionality adaptation. However, there are a few features in our model that make selection not a trivial task. Instead of using a simple and direct approach in comparing and selecting facets for functionality adaptation, we base on the selection approach and propose a *two-phase adaptation*.

4.5.1 Adaptation Challenges

In order to be usable in pervasive computing, our model is designed to be quite dynamic. It is this dynamism that makes selection not a trivial task:

1. **Dynamic Configuration.** Usually, when a component is designed to require other components for providing a certain service, it knows, at compile-time, the exact components that are needed. Therefore, it is possible for a service to have several fixed configurations of service components for selection. However, this static binding is not flexible for updating a service with new components. Instead of having a fixed binding between the components, our model uses a dynamic one. Facets that constitute a service are dynamically bound when the corresponding functionalities are requested at run-time. This dynamic binding creates the flexibility for dynamic update of services. However, it also causes the facet execution tree unable to be known at compile-time. We are not even able to have any idea about the calling

depth or the complexity (i.e. the number of intermediate nodes) of the tree. All these uncertainties due to the dynamisms make it impossible for a service to have any fixed configuration of components for selection.

2. **Dynamic Resource Usage.** Unlike web contents whose resource usage can be determined statically, codes have dynamic behavior. Given a piece of code, different executions of the code may yield different results in the dynamic resource usages. For example, an image decoder uses 1MB of memory resources to decode a 800×600 image of 8-bit color depth; and 2MB when the image to be decoded is of size 1024×768 with a color depth of 16. This dynamic behavior of a code's execution makes it difficult to determine the amount of resources that would be used when it is to be executed in a heterogeneity of client devices.

For the case of our proxy system, the unpredictability of the facet execution trees and the dynamic resource usages of the facets pose a great difficulty to the proxy system in determining whether a functionality provided by an unknown sequence of facets can be completed in the client device. This decision is to be made before the root facet contributing to the functionality is selected for execution.

4.5.2 The Idea

Before going into the details of how these challenges can be overcome, an overall idea of the two-phase adaptation and how the proxy system uses it to achieve functionality adaptation are given.

Figure 4-1 shows the processes involved in a two-phase adaptation. The aim of the two-phase adaptation is to select a facet from the available ones, and that is considered to be the best-suited to the client. The first phase, called the *filtering phase*, is to filter the facets that satisfy the requirements of the client. These requirements include, at least, the functionality needed by the client and the amount of resources available in the client device for executing the specified functionality. All the facets filtered by the first phase have satisfied the client's requirements and are eligible for further processing. The second phase, called the *selection phase*, is to select a facet that best-suits the device user. This decision is based on the user preferences and other execution contexts of the client. The facet resulting from the two-phase adaptation is considered functionality-adapted and returned to the client.

There are three key techniques being used in the two-phase adaptation, namely *functionality filtering*, *resource filtering*, and *context selection*. They are important to the adaptation of the service

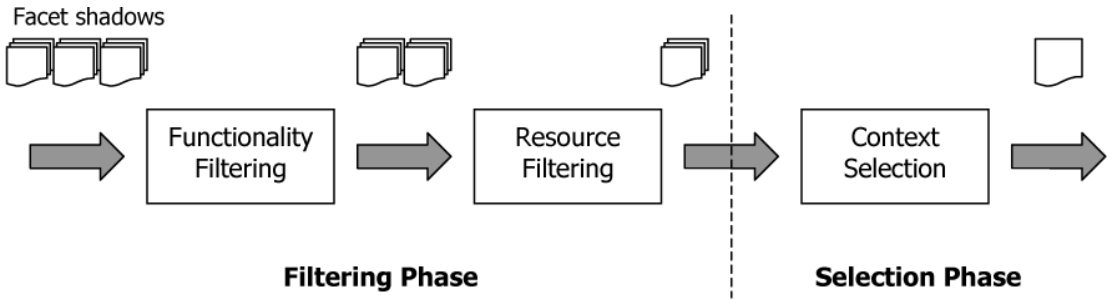


Figure 4-1: The processes in a two-phase adaptation.

codes and are discussed in separate subsections and chapters.

4.5.3 Functionality Filtering

Being used in the first of the filtering phase, its main duty is to filter facets that satisfy the specified functionality. In order for a facet suitable to be returned to the client for execution, the first, and the most important, criterion to satisfy is the functionality. Facets that, when executed, cannot achieve the functionality required by the client are useless and should be filtered off. Using a functionality filter here, therefore, has the purpose of cutting down a significant number of facets to be processed.

If the functionality filter is only capable of filtering facets with exactly the same functionality as that specified by a client, the requirement is too strict. It is possible to have facets of a similar functionality, but can achieve the functionality required by the client. These facets are usually more powerful (i.e. of a greater capability), and can achieve more on top of the client-specified functionality. To have a better understanding on this, let us take a look at an example. A client requires a facet that can decode an image of size no more than 800×600. A facet F_{decode_1}, with exactly the same functionality, is filtered without doubt. After some careful analysis, facets F_{decode_2} and F_{decode_3}, responsible for decoding images of size no more than 1024×768 and 1280×1024 respectively, should also be able to satisfy the functionality requirement of the client. They are considered more powerful than facet F_{decode_1} because they can handle images of a larger size. These facets, of a similar functionality but capable of achieving the desired functionality, are said to have *compatible functionalities* and should also be filtered. This leads to the design of the functionality filter in our proxy system in that it does not only filter facets with exactly the same functionality, but also with compatible functionalities.

In order to filter facets of compatible functionalities, we need to define "functionality compati-

Facet F_{decode_1} *Description:* Image decoding *Input:* A file containing the image to be decoded *Output:* An array containing the decoded image *Pre-conditions:* The size of the image to be decoded $\leq 800 \times 600$ *Post-conditions:* The image is decoded and put in the array
Facet F_{decode_2} *Description:* Image decoding *Input:* A file containing the image to be decoded *Output:* An array containing the decoded image *Pre-conditions:* The size of the image to be decoded $\leq 1024 \times 768$ *Post-conditions:* The image is decoded and put in the array
Facet F_{decode_3} *Description:* Image decoding *Input:* A file containing the image to be decoded *Output:* An array containing the decoded image *Pre-conditions:* The size of the image to be decoded $\leq 1280 \times 1024$ *Post-conditions:* The image is decoded and put in the array

Table 4.1: The functionalities of facets F_{decode_1}, F_{decode_2} and F_{decode_3}.

bility". That is, from the above example, how can we know that facets F_{decode_2} and F_{decode_3} have compatible functionalities as facet F_{decode_1}? In order to find out the answer, we try to compare the functionalities of these facets. As mentioned in Chapter 3, the functionality of a facet is specified by its description, input, output, pre-conditions and post-conditions. Using this definition, we write out the functionalities of the facets in Table 4.1.

From the table, it can be seen that although the descriptions, inputs, and outputs are the same, the three functionalities are considered to be different due to the variations in the pre-conditions. These pre-conditions, in fact, specify the *capabilities* of the facets (i.e. the abilities of the facets in performing their functionalities). The three facets are all used for decoding images, but have different capabilities regarding how large an image can be handled. F_{decode_1} is capable of decoding images of sizes no larger than 800×600; F_{decode_2} is capable of decoding images of a larger size, but not larger than 1024×768; while F_{decode_3} can handle images of the largest size, up to 1280×1024. Facets F_{decode_2} and F_{decode_3} are, therefore, able to work with all the images that F_{decode_1} can handle. These facets, that can also achieve the client-specified functionality but of a greater capability, are what we are looking for in defining functionality capability.

Therefore, if we consider the pre-conditions to specify the capability of a facet by a range of values that it can handle, facets that have the same descriptions, inputs, and outputs, but a pre-

condition of a larger range are what we are looking for. We call the range of values specified in the pre-condition the ***capability range***. In order to formulate a method for identifying a greater capability range, we need to use our knowledge in inequalities. Consider two simple ranges:

$5 < x < 10$

$0 < x < 15$

The range $0 < x < 15$ includes all the values in the other range ($5 < x < 10$). We can get the greater range, i.e. $0 < x < 15$, by using the *OR* operator in inequalities. Therefore, if the pre-conditions are simple ranges in one dimension, we can identify whether a facet can achieve the client-specified functionality by: first, comparing the descriptions, inputs, and outputs; and second, compare the pre-conditions (in fact, the capability ranges) if the descriptions, inputs, and outputs are the same. We call this process the ***capability matching***. If after applying the *OR* operation to the capability ranges, the resulting range is the capability range of the facet, then the facet can achieve the client-specified functionality.

For most of the time, the capability ranges are not only in one dimension. They usually involve a number of dimensions, with each dimension specified by a range. In that case, finding a greater capability range requires applying the same method to the n-dimensions. Consider the following two-dimensional ranges:

$5 < x < 10$ and $y < 20$

$0 < x < 15$ and $y < 30$

In order for a facet to be considered as having a greater capability, it should have a larger capability range for every dimension. Therefore, we apply the *OR* operation to each dimension, if it yields back the capability range of the facet, the facet is considered to have a greater capability and be able to achieve the client-specified functionality. So, in the above two-dimensional ranges, if "$0 < x < 15$ and $y < 30$" is the capability range of a facet, and the other is extracted from the client-specified functionality, applying the *OR* operation to each dimension yields back the same capability range, implying that the facet has a greater capability.

Actually, only considering the pre-conditions for the capability of a code is not enough. Although the pre-conditions already specify the capability range the code can handle, it does not involve any output capability (i.e. the range of outputs that can be produced) that might affect the ability of the client to present the resulting contents. For example, a facet that can produce images of sizes not less than 1024×768 is not be able to produce an image of a smaller size, e.g. 800×600,

that suits the client device's screen size. Therefore, besides considering the pre-conditions, the post-conditions also need to be considered as well, when filtering facets of a greater capability. From the example just mentioned, it seems that we need a smaller range for the post-conditions. However, this is only true regarding the resulting contents for displaying in the resource-constrained devices. Facets that produce images of a smaller size can also be filtered. For other cases, where the output is not for presentation in the client devices, getting a smaller range might not be correct. Consider the following example. Facet F_{passwd_1} can produce passwords of length no more than 8, while facet F_{passwd_2} can produce passwords of length no more than 5. If the client requires a facet that can produce passwords of length at most 6, F_{passwd_1} should be filtered instead of F_{passwd_2} in order to achieve what the client requires. That is, a larger range (password length ≥ 8) is required. Therefore, there is no general method if we want to take the post-conditions into account when considering compatible functionality. To keep it simple, we require the post-conditions to be the same if the functionalities are compatible. This requirement can, at most, cause some facets having compatible functionalities not able to be filtered. It can only reduce the number of facets that can be filtered, but cannot result in facets having incompatible functionalities such that they cannot achieve what the client requires.

To summarize, compatible functionality can be defined as a functionality that has:

- a similar functionality as the one specified by the client (i.e. the same descriptions, inputs, and outputs), and
- a capability that can achieve the functionality specified by the client (i.e. a same or greater capability range specified in the pre-conditions, and a "suitable" range for the post-conditions).

Note that a compatible functionality also includes functionalities that are exactly the same.

With the concept of compatible functionality, the functionality filter is able to filter facets that can achieve the functionality specified by the client, and the rest are filtered off. The filter can be seen as a predicate:

```
similar functionality && same or ‘‘greater’’ capability
```

Facets that are filtered (by satisfying the predicate) can then be passed on to the next filter to see if they can be able to provide the functionality in the client device.

4.5.4 Resource Filtering

With the facets satisfying the functionality of the client in functionality filtering, resource filter ensures that facets being filtered satisfy the resource requirement of the client device. This resource requirement is the amount of available resources for executing the specified functionality in the client device. The resource filter is, thus, responsible for filtering facets such that the functionalities they provide can be completed in the client device.

In order to filter these facets, the proxy system needs to be able to know the resource usages of the functionalities provided by each of these facets, so that it can compare them with the available resources for the execution in making the filtering decision. Recall that a functionality is not necessarily provided by a single facet. It can be provided by executing a sequence of facets. Therefore, the resource usage of the functionality provided by a facet is not only the resource usage of the facet, but also depends on the resource usages of all the facets being used at run-time that help to fulfill the functionality. This implies that the facet execution tree is needed to calculate the resource usage of a functionality. However, this is some future information that cannot be known until run-time. Resource filtering, therefore, requires some prediction of the facet execution tree for calculating the resource usage of a functionality. Details of the prediction that helps to decide whether a functionality can be completely provided in the client device are given in the next chapter.

After resource filtering, the filtering phase is said to be completed. Facets being filtered should satisfy all the requirements specified by the client, such as the functionality requirement and the resource requirement. They should be able to achieve the desired functionality, execute and complete in the client device, and are *candidates* to be selected in the selection phase.

4.5.5 Context Selection

In the last of the two-phase adaptation, the proxy system selects a facet from the candidate facets that have passed the filtering phase. Normally, the selection phase can be thought of having a number of selection passes (c.f. the filters in the filtering phase), each responsible for a different selection criterion. However, the selection criteria are usually dependent on the device user. This makes it difficult to generalize a few selection criteria that are common to all device users and, at the same time, critical to the selection (c.f. the functionality and resource filters used in the filtering phase). Therefore, for the sake of our convenience, we group them together and consider it as a single pass, which is called the context selection.

Unlike the filtering phase that filters facets satisfying the requirements, the selection phase selects the one that best-suits the client. This selection cannot result in a facet unable to perform the functionality in the client, but only how suitable it is for the client. In determining the suitability, the satisfiability of the device user is the main concern. The proxy system, therefore, needs some information about the device user in order to make a good decision that reflects the user's desire. This information that reflects the user's desire is the ***user preferences***. It is a list of properties that the returned facets are preferred to have, and is usually stored in the *user profile*.

With this information available, selection is mainly base on the user preferences. These user preferences can indicate any properties of a facet (i.e. any information appearing in a facet shadow), such as vendor, version, resource usage, etc. For properties like "vendor" and "version", it is easy to understand them as user preferences. But this is not the case for "resource usage". Allowing users to specify the preferred resource usage of a facet seems to be strange, as this should not be their concerns. As a user, they only care whether the facet can perform the desired functionality in the device, but not how much dynamic resources a facet actually uses during execution. In fact, a resource usage level is expected for the "resource usage: in the user preferences. This resource usage level indicates the *relative resource usage* among all the candidate facets for selection. Note that this only considers the resource usage of a single facet, and does not involve the resource usages of its dependencies. For example, a resource usage level of "high" indicates that the facet with the largest amount of resource usage, among all the candidate facets, is preferred. A resource usage level of "low" is just the other way round. A resource usage level of "medium" takes the one with medium resource usage among the candidates. By allowing users to specify the resource usage level, they can control the relative resource usages of the facets that constitute a service. For example, a user would like to use more resources for security functions, and less for other kinds of functions. If the user asks for a complete e-mail service, which requires a facet for composing the contents of the e-mail, and another for sending. Since the content to be sent is required to go through the network, the facet used for sending the content, in turn, requires a facet for encoding the contents before the actual sending. Figure 4-2 shows the corresponding facet execution tree. The facet for encoding (F_{encode_1}) is, no doubt, a security function, and the user prefers it to use more resources than the facet for sending (F_{send_1}). By using the resource usage level information from the user preferences, the proxy can choose from among the candidate facets $F_{send_1}, F_{send_2}, \ldots$ the one that uses the smallest amount of resources, leaving comparatively more resources for the encoding facet that would be used later by the sending facet.

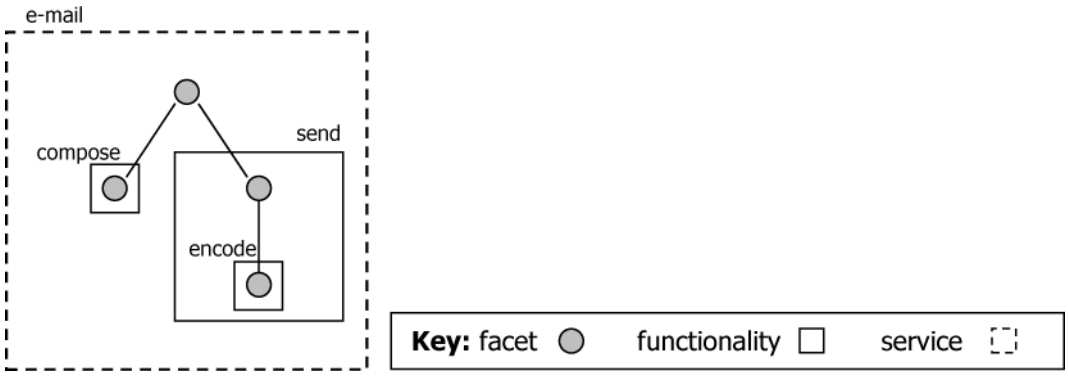


Figure 4-2: A facet execution tree for an e-mail service.

Apart from the user preferences, the proxy system can also make use of other information for better selection. This information should help the proxy system to select a facet suitable for the user. Examples include the status of the personal proxy cache, and the user's past usage pattern. They are information that can be stored in the proxy system and used as a kind of *proxy preferences* for selection. If a facet has been cached in the proxy, the time for returning it to the client can be less. This leads to a better performance that is likely to mean a higher preference for selection. In another case, if the usage pattern of a user indicates that a facet has been used before. It is also likely that the facet has a higher preference than the other candidates because they have ever satisfied the client (in another context). Actually, whether these facets are suitable to be given a higher preferences depend on the proxy system. But these information might be able to act as some proxy preferences and help the proxy system in deciding the facet that is better for the client.

Since the user preferences and the proxy preferences are used extensively for the selection, the proxies need to be able to determine which facet is more suitable if there are more than one facet satisfying the user/proxy preferences. Consider a facet that satisfies the user preferred vendor, and another facet that satisfies the user preferred version. How can the proxy system decide which is more suitable to be selected? In order to help the proxy system in making this decision, a *scoring system* is needed. Scores are given if a preference is satisfied. Each preference item (e.g. vendor, version), has a different score, depending on their importance to the user. The user can assign a score to each user preference item in the user profile. Figure 4-3 shows an example. If an item is satisfied by a facet, the corresponding score is added to the facet. Similarly, the proxy system can assign a score to each proxy preference item, e.g. cache status, past usage pattern. The corresponding score is added to the facet if it satisfies a proxy preference. After calculating the scores, the facet that

Vendor: Peter Ng (10) Version: 1.0 (5) Resource Usage: High (12)

Figure 4-3: An example of user preferences in a user profile.

scores the highest should be best suited to the client, and is, thus, selected and returned to the client.

After selection, the facet should be able to satisfy both the client device, and the device user. All the decisions made are based on the run-time information available at the moment of request, such as the available resources in the client device, the required functionality, and the user preferences that depends on the environment, all subjected to change in execution contexts. Therefore, the facet returned should be best suited to the client, according to the run-time execution context at the time of request.

4.5.6 Functionality Prediction

The two-phase adaptation is base on the fact that the functionality required is provided by the client. The proxy system needs this information in order to adapt a suitable facet for providing the desired service. This information is only known by the proxy at run-time when the client requests a functionality, making it impossible for the proxy system to pre-adapt a facet for the client. Instead of waiting for the client to request for a functionality, it would be better if the proxy system is able to predict the functionality that would be used in the near future, according to the client's execution context; and select a facet accordingly. However, there are a few problems with this prediction:

- **Bandwidth Limitation.** Bandwidth limitation is a major problem in small mobile devices, due to its mobile nature across the network. The bandwidth of the wireless technology, being at about 2Mbps, is much less than the bandwidth of a fixed network, which can be about 100Mbps (or even 1Gbps). No matter how fast the wireless technology improves, there must be a gap between the two. This makes the bandwidth in small mobile devices a precious resources, and it would be wasted if the prediction is incorrect. Therefore, in our model, the proxy system needs to take this issue into account. In order to reduce the time for adapting a suitable facet at run-time, the proxy system tries to predict the functionality required, and selects a facet according to the predicted functionality. However, this time, instead of returning the predicted facets to the clients, it only stores the facets in the user's personal proxy cache. A facet is returned to the client only upon request, in order to minimize the waste in band-

width due to mis-prediction. Although the pre-adapted facet is not returned to the client in advance, pre-adaptation helps improving the access latency if the prediction is correct. This improvement in access latency is significant especially in the initial requests after the client moves to another location and continuing the previous computing session.

- **Prediction Difficulty.** Another problem in predicting a functionality is the difficulty in detecting a change in functionality if there is no major change in the client's execution context. A minor change in execution context (e.g. time advanced by a minute), might not cause any change in the functionality required and is not essential to our prediction. Therefore, in our model, we assume that functionality changes only occur when there is a major change in a client's execution context, such as a change in location; and prediction is only done by then, due to a possible change in user activity.

Base on the above assumptions, the proxy system can make minor predictions of the functionalities to be requested. User behaviors, as mentioned before, are likely to be different in different execution contexts. This implies that the *sets of services* required in different contexts are likely to be different. Despite that, we still believe a subset of these services in different execution contexts can be the same [20]. In order to illustrate this idea, consider this example. A mobile-user, reading while he is waiting for a bus, might require services like a reader, a dictionary, and a marker for book-marking the paragraph he left off. When he got back to office for a meeting (i.e. a change in location), he might also need a reader for reading the meeting agenda, a marker for reminding him of things being discussed, and an e-mail function for sending a drafted minutes to the absentees. Although the sets of services in the two locations are different due to the different activities (reading and having a meeting), there can be a subset (reader and marker) that is the same.

Simply knowing there exists a possibility that some services that have been used might also be used in the new location does not help much. However, if there is some *correlation* among these services, it is possible for the proxy system to have minor predictions of the services to be requested. In the previous example, the reader and the marker functions are correlated in the sense that one is used after the other. In both situations, the reader function is used before the marker function. If this kind of correlations between the services is analyzed, they can help predicting the functionalities to be used. So, if the correlation between the reader and the marker functions is analyzed before the user got back to his office, the proxy system should be able to predict a future request of a marker function once the client requests for a reader function. However, just a single occurrence of the

marker function after the reader function is not enough to be considered as a correlation between the two. In order to identify the correlations between the services, the proxy system can analyze the probability that a service is used after another. Two services are only considered to be correlated if the probability of using one after another is high, e.g. more than 3 times. This correlation between services helps the proxy system in predicting the functionalities to be used. It can then pre-adapt facets for the clients, and store them in the proxy cache for reducing access latencies.

4.6 Summary

This chapter gives a detailed discussion about functionality adaptation: the kind of adaptation that the proxy system is required to provide in pervasive computing environments, the needs of functionality adaptation, and the kind of contexts that affect the adaptation decision. We have also pointed out the difficulties of this adaptation, where the main reason is due to the dynamic composition of services and the dynamic behaviors of the service codes, which are different from the existing adaptations. Lastly, an overall picture of how our proxy system adapts the functionality using a two-phase adaptation is given.

The three key techniques used in the two-phase adaptation helps adapting the functionalities to the clients. Functionality filtering ensures facets to achieve the functionality requirement of the client. Resource filtering ensures the functionality provided can be completed in the device; and context selection selects a facet that best-suits the user and other execution contexts of the client.

Having an overall picture of the adaptation introduced in this chapter makes it easier to understand the details of the core mechanism — resource filtering — which will be given in the next chapter. It helps the proxy system to ensure the facets returned can be executed and completed in the resource-constrained devices.

Chapter 5

Conservative Prediction

Having grasped the concept of functionality adaptation and how our proxy system deals with it by a two-phase adaptation, this chapter gives the details of resource filtering used in the first of the two-phase adaptation. In order to filter facets that satisfy the resource requirement of a client device, information about the dynamic resource usage is needed. Being able to predict this dynamic resource usage is, therefore, important in resource filtering. This chapter first describes the challenges of resource filtering, and discusses why resource usage prediction is needed. Then, it goes into the details of the prediction and illustrates how the prediction can be achieved with our conservative prediction. Lastly, an analysis of the conservative prediction is given to see how good it is in functionality adaptation, and the chapter ends with a brief summary.

5.1 Challenges of Resource Filtering

The main purpose of resource filtering in the first of the two-phase adaptation is to filter facets that satisfy the resource requirement of a client device. At the first glance, it seems that this filtering process is trivial and only involve simple comparison (by comparing the resource usage of a facet with the resources available in the client device for execution). In fact, this is only an illusion. There are a few challenges in making this filtering decision. The fact that facets are mobile code components and can be dynamically composed brings out these challenges.

5.1.1 Dynamic Configuration

In our model, facets can be dynamically composed to form services. In order to fulfill the functionality that a facet aims to provide, it can call upon other facets to help fulfilling parts of its functionality.

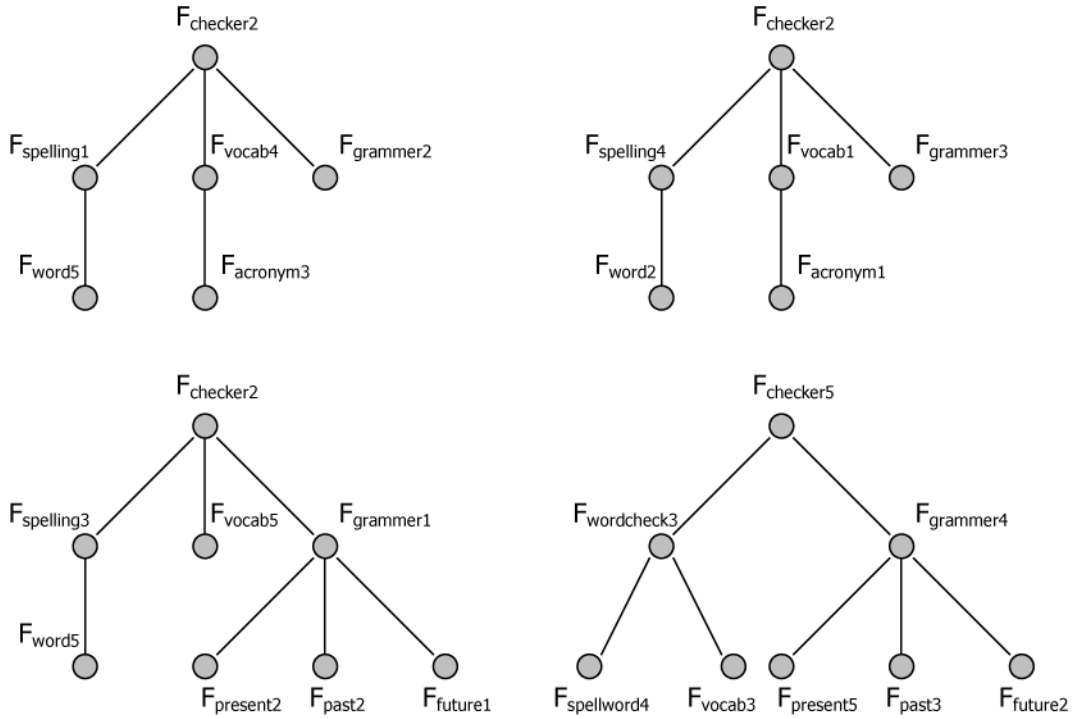


Figure 5-1: A number of possible facet execution trees, providing the same functionality.

For example, a facet for sending an e-mail (F_{email_2}) may require a facet for encrypting the e-mail contents before sending. That means, two facets are needed together to fulfill this e-mail sending functionality. However, these facets are not statically bound to form an e-mail service. Any facet that can achieve the functionality of "encrypting the e-mail contents" can be used by F_{email_2} in providing the e-mail service. The actual facet to be used by F_{email_2} for encrypting its contents is dynamically bound, and not known until run-time. This dynamism allows a suitable facet to be selected flexibly at run-time, but also makes it difficult to know ahead of time the actual facets to be used (i.e. the facet execution tree).

Figure 5-1 shows a number of possible facet execution trees to illustrate the difficulty for the proxy system to know an actual execution tree used by the client at run-time. These possible execution trees arise from the fact that the actual facets to be used for fulfilling a functionality are selected at run-time. Different run-time execution contexts result in different facets being selected, and each of these facets has a different implementation and may require different functionalities to help fulfilling their own functionality. All these variations result in a number of possible execution trees that a facet can take. The number of possible facet execution trees grows with the number of dependencies a facet has, and the number of candidate facets that satisfies a dependency. Let us take

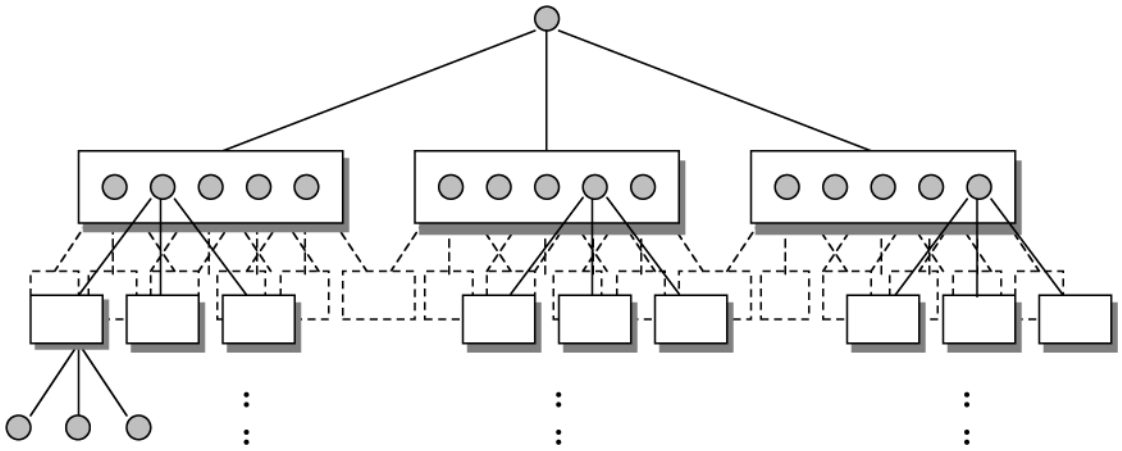


Figure 5-2: Representing a number of possible facet execution trees.

a look at an example to have a concrete idea how large this number can be. Assume, on average, that a facet has three dependencies, and each dependency has five candidate facets available for selection. On top of this, we also assume the facet execution tree is a complete tree, with a calling depth of three (see Figure 5-2). The total number of possible facet execution trees can then be estimated:

$$((5^3 \times 5)^3 \times 5)^3 = 5^{3^3+3^2+3^1} = 5^{39}$$

This, in general, implies the number of possible facet execution trees can be estimated by the formula:

$$c^{d^h+d^{h-1}+d^{h-2}+\cdots+d}$$

where c is the number of candidate facets for each dependency, d is the number of dependencies a facet has, and h is the calling depth of the execution tree.

The above formula shows that the number of possible facet execution tree is a very large number. In fact, this is only a rough estimation, as the number of dependencies differs for each facet, and the number of candidate facets available for selection differs as well. Despite that, the formula is already enough to illustrate that with all these variations, there can be many execution trees that a facet can take. This is why it is difficult for the proxy system to know the facet execution tree that represents the facets to be used for achieving a functionality, which is only available at runtime when all of the facets involved have been selected. Without knowing the actual execution tree for providing the functionality to the client, it is difficult to determine the resource usage for executing the functionality. It is not only difficult to determine the actual facet execution tree, but

the proxy system does not even have any idea about the calling depth or the complexity (number of intermediate nodes) of the tree, as they all depend on the facets being selected along the way during execution.

5.1.2 Dynamic Resource Usage

Unlike Web documents, facets are code components instead of data contents. The difference between the two is that data is usually static and consumes a fixed amount of resources, while codes can be dynamic and the amount of resources they consume varies with the executions. This dynamic resource usage is due to the dynamic behavior of codes, where the execution behavior for an execution is different from another. This difference between code and data is important if code execution is required to be performed in the resource-constrained devices, which is desired for pervasive computing. Data contents with static resource usages can be safely downloaded to the client devices if they satisfy the resource requirements of the devices. But this is not the case for codes. Mobile codes that can be downloaded to a client device do not imply they can be executed in the device. The device may not have enough resources for its execution. The amount of resources that a code consumes during its execution (i.e. the dynamic resource usage) has to be taken into account when code is downloaded to the client, which is not an easy task.

The dynamic resource usage that a code consumes during execution cannot be known ahead of time. This is because this dynamic resource usage depends on the executions, and is not simply a static data that can be known at compile-time. A code, with the use of some control structures (e.g. if-then-else, while, ...), may have different possible execution paths. Different executions may cause these different execution paths to be gone through, which are likely to result in different resource usages. Even worse, the dynamic resource usage of a code does not only depend on the structure of the code itself. It also depends on the size of the input provided by the client, and cannot be known until run-time. For example, a code for encoding takes an image as input, and depends on the size of the image to allocate enough memory for processing. Different sizes of an image, therefore, results in different amounts of memory being allocated; and the actual amount cannot be known unless the client provides an image to the proxy at run-time, but this is not recommended in terms of privacy concern. All these uncertainties causes the dynamic resource usage of a code unable to be known ahead of time.

Since computing devices have limited resources, especially for the case of small mobile devices, not knowing the dynamic resource usage of a piece of code certainly places a difficulty in

determining whether the code can be executed and completed in the resource-constrained device. This problem needs to be solved in order to enable codes to be downloaded for execution in pervasive devices.

5.1.3 Combining the Dynamics

In fact, the above dynamics affect the decision of whether a service (or a functionality) can be completed in the client device. During the process of resource filtering, a number of facets need to be checked to see if they satisfy the resource requirement of the client device. In order for a facet to pass this checking, the functionality it provides should be able to execute and complete in the client device. This depends on the facet execution tree that represents the facets to be used at run-time, and the dynamic resource usages of the facets in the execution tree. The uncertainty of these information, therefore, places a difficulty in resource filtering. All these challenges have to be overcome, such that functionality adaptation can be achieved and the facet selected by the proxy system should be able to provide the desired functionality in the client device. The client device should not be able to start executing a functionality if it does not have enough resources for its completion.

5.2 The Problem Formulation

In order for the proxy system to be suitable for pervasive computing, it has to take into account the heterogeneity of the client devices. That means, the facets being selected have to be adapted to the client devices. Resource filtering is, therefore, needed to ensure the facets returned to the clients can have their functionalities completed in the devices. In resource filtering, facets satisfying the functionality requirement are checked in turn, and those that can complete the functionality in the client device are filtered.

This filtering process, in fact, can be formulated as: Given a resource R, and a list of facets $F_1, F_2, \ldots, F_i$ satisfying the required functionality. We need to filter all the facets F_j where $j \leq i$, such that the resource usage for F_j to complete the functionality, $res(T_F(j))$, does not exceed the available resources; i.e. $res(T_F(j)) \leq R$.

In order to determine whether a functionality provided by a facet can be completed in the client device, we need to be able to know the dynamic resource usage in executing a functionality, i.e. $res(T_F(j))$. If this resource usage does not exceed the amount of available resources in the client

```
Input: gif-image
Output: jpeg-image

bitmap ← gifdecoder(gif-image);
jpeg-image ← jpegencoder(bitmap);
return jpeg-image;
```

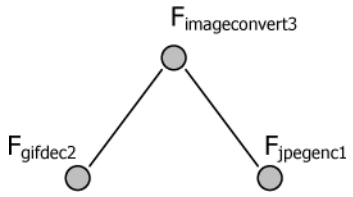


Figure 5-3: The psuedocode of an image converter, and the corresponding facet execution tree.

device, then the facet can provide the desired functionality in the device and can be filtered as a candidate for selection. The dynamic resource usage for a facet to complete its functionality does not only concerns the resource usage of the facet, but also all the facets that are required to help fulfilling the functionality, which are dynamically bound at the time of request. These facets are not known when the selection of a facet at a higher level in the facet execution tree is made. For better understanding, let us look at an example. Consider an image converter that can convert an image from a gif format to a jpeg format. The image converter might, first, take a gif image and ask a gif decoder for decoding the image to a bitmap. It can then take the bitmap to ask a jpeg encoder for encoding it to a jpeg image to be returned. The corresponding code segment of the image converter and a possible facet execution tree are shown in Figure 5-3. It can be seen that the resource usage for converting the image depends on the resource usage of the image converter, the gif decoder and the jpeg encoder, because they all work together in providing the functionality of converting an image from gif to jpeg format. And, in general, a facet can have more than two dependencies, and the facets used for satisfying these dependencies may also require other facets to help fulfilling their functionalities. The resource usage of the functionality provided by a facet, thus, depends on the resource usage of itself, as well as the resource usage of its *subtrees* in the facet execution tree; i.e.

$$res(T_F(j)) = f(res(F(j)), res(T_{F_1(j)}), res(T_{F_2(j)}), \ldots$$

where $T_{F_1(j)}, T_{F_2(j)}, \ldots$ are the subtrees of $T_F(j)$.

Knowing how the resource usage of the functionality provided by a facet can be calculated helps resource filtering. But, the problem is that, we do not know the resource usage of its subtrees, as the actual facet execution tree cannot be known until run-time. This is why prediction is needed. The proxy system needs to predict the dynamic resource usage of its subtrees so as to determine whether a facet can provide the functionality in the client device. Let us reconsider the above example and focus on the prediction process. Assume that the proxy system has received a request for an image

converter that can convert a gif image to a jpeg image. In order to return a suitable facet that does the conversion, the proxy system first filters facets that can achieve the specified functionality, then see if their functionalities can be completed in the client device. During this resource filtering process, the image converter facets are checked. Consider one of these image converters, say $F_{imageconvert}$ in Figure 5-3, that requires a gif decoder and a jpeg encoder. The resource usage for executing the functionality can then be calculated by first calculating the resource usage of this facet, i.e. $res(F_{imageconvert})$. This resource usage can be estimated, since more concrete information of the input should be available to the proxy system when this image converting functionality is requested. After that, the resource usages of its dependencies are needed. They correspond to future requests and are not known at this moment. Therefore, the proxy system has to predict these future requests and the facets that would be selected. In this case, the proxy system might predict the facets F_{gifdec} and $F_{jpegenc}$ shown in Figure 5-3 to be used for performing the gif decoding and jpeg encoding functionalities respectively. Only with these predicted facets can the proxy system estimate the dynamic resource usage for executing the image converting functionality.

In general, calculating the resource usage for executing a functionality provided by a facet requires the proxy system to predict the future resource usage of the functionality, which in turn requires:

- *prediction of future requests* that are needed to help fulfilling the functionality,
- *prediction of actual facets* (and eventually the facet execution tree) that will be selected, base on the predicted requests,
- *prediction of the resource usage* for executing the whole functionality, base on the predicted facet execution tree.

However, without knowing the actual run-time information, all these predictions are not accurate. These inaccuracies are due to the dynamic behavior of the codes during execution, the dependence of the resource usage to the input, and the selection of a facet according to the run-time execution context. Inaccurate predictions, sometimes, may result in the requested service not able to be completed in the client device. For example, if the image converter returned to the client is predicted to use 480KB memory to provide the functionality, which should be able to be executed and completed in the client device that has 500KB memory available for the execution. But, at run-time, it actually uses 520KB memory, which cannot be provided by the client device. The service, thus, might be stopped in the middle of the execution and unable to be completed.

Actually, inaccuracy is not always a concern. If, in the above example, the service uses 495KB memory at run-time instead of 520KB, the image converter returned (although with the resource usage predicted wrongly), is still able to execute and complete the functionality in the client device that has 500KB memory available for the execution. That means, the resource usage predicted does not need to be 100% accurate. The aim of resource filtering is only to filter facets whose functionality can be executed and completed in the client device, and not predicting the facet execution tree that the client uses. The prediction of the execution tree is only to help calculating the resource usage for executing the functionality. An inaccurate facet execution tree is, therefore, still acceptable if the functionality represented by the execution tree can be completed in the client device. So, instead of focusing on predicting an accurate facet execution tree, and hence the resource usage, we change our focus to answer the following question:

> *Is it possible to predict a resource usage for executing the functionality without aiming at an accurate facet execution tree, and whose inaccuracy cannot result in the service not being able to be completed in the client device, if the predicted resource usage can satisfy the resource requirement of the device?*

In fact, the prediction problem can be formulated more precisely as: Given a facet F and the client's available resource R. We need to find a facet execution tree T for the facet, with facets $F, F_{T_1}, F_{T_2}, \ldots, F_{T_{i-1}}$ where i is the number of facets in T and F being the root facet, of resource usage $res(T)$, such that:

1. $res(T') \leq res(T)$, where T' is the facet execution tree used at run-time, with facets $F, F_{T'_1}, F_{T'_2}, \ldots, F_{T'_{j-1}}$ where j is the number of facets in T', F is the root facet, and $res(T')$ is the resource usage of T', and
2. $res(T') \leq R$ iff $res(T) \leq R$.

If such an execution tree exists, then the facet F satisfies the resource requirement of the device, and can be a candidate for selection.

After defining the problem, we are now left with a few questions:

- how to predict such a resource usage for executing the functionality (i.e. $res(T)$ for any facet execution tree T)?
- is the above a good prediction? Would any inaccurate prediction cause the service unable to be completed in the client device (i.e. will $res(T') \geq R$ but $res(T) \leq R$)?

5.3 Conservative Prediction

This section tries to tackle the question: how to have a reasonable prediction of the resource usage for executing a functionality without aiming at an accurate facet execution tree? This prediction needs to be reasonable so that it almost reflects the actual resource usage at run-time. But before looking at the prediction, we first need to know how a predicted resource usage for executing a functionality is said to satisfy the resource requirement of a client device. This is needed because the final goal of the prediction is to determine whether the functionality can be completed in the client device.

5.3.1 Satisfying the Resource Requirement

As mentioned, resource filtering is to filter facets whose functionality can satisfy the resource requirement of the client device. In order to determine this, the dynamic resource usage for executing a functionality needs to be calculated. This resource usage depends on the facets being used at run-time to help fulfilling the functionality, and therefore base on a facet execution tree. So, given a facet execution tree T, we need to know how to calculate the resource usage for executing the functionality that is represented by T, i.e. $res(T)$.

In the usual case, the resource usage of a tree can be calculated by summing the resource usage of the root facet and the resource usages of all its subtrees. Assuming that the facet execution tree T is rooted at a facet F, and has i subtrees: $T_1, T_2, \ldots, T_i$. The resource usage can be calculated by:

$$res(T) = res(F) + res(T_1) + res(T_2) + \cdots + res(T_i)$$

The resource usage of a subtree, e.g. $res(T_i)$, is calculated in the same way, which eventually implies summing the resource usages of each facet in the facet execution tree. And, the functionality is said to satisfy the device's resource requirement iff:

$$res(T) = res(F) + res(T_1) + res(T_2) + \cdots + res(T_i) \leq R$$

However, our proxy system is based on Sparkle, where client devices have the capabilities of discarding facets after being used. That is, when the facet has completed its execution, it can be discarded to free up the memory in the device for other facets to be executed. In this case, only facets that are in use (i.e. whose functionality has not been completed) occupy memory. For example, the

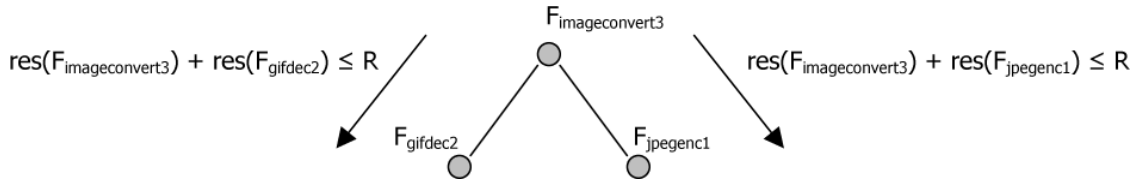


Figure 5-4: The image converter satisfies the resource requirement of the client.

image converter discussed before requires two facets: a gif decoder and a jpeg encoder. When the image converter facet is executed, it requires to decode a gif image, and a gif decoder is then on-demand downloaded to the client device for decoding. During the decoding of the gif image, the image converter facet is still considered in use, since its functionality in achieving the gif to jpeg conversion has not been completed. Therefore, both the image converter and the gif decoder facets are in use and occupy memory in the client device. When the decoding has completed, the image converter continues its execution and finds that it now requires a jpeg encoder for encoding. The gif decoder, which has already completed its functionality, can now be discarded; and a jpeg decoder can be downloaded. The memory is, again, occupied by two facets; but this time the gif decoder is replaced by the jpeg encoder. These two facets can also be discarded after they have completed their functionalities. Figure 5-4 shows the image converter satisfying the resource requirement.

In the above example, it can be seen that the gif decoder and the jpeg encoder will not be in use at the same time. When the gif decoder is in use, the jpeg encoder is not yet needed; when the jpeg encoder is in use, the gif decoder has been discarded. That means, they are not occupying the memory at the same time, and their resource usages do not need to be added up for the resource usage required to execute the functionality of "gif to jpeg conversion". In general, the dependencies of a facet are used one after another, and the amount of resources required for executing all its dependencies depends only on the *maximum* resource usage of its dependencies. Therefore, the resource usage for executing the functionality represented by a facet execution tree T, in this case, is calculated by summing the resource usage of the root facet and the maximum resource usage required by its subtrees; i.e.

$$res(T) = res(F) + max(res(T_1), res(T_2), \ldots, res(T_i))$$

And, the resource requirement is satisfied if the total resource usage involving **any of its dependen-**

cies cannot be greater than the available resource usage; i.e.

$$res(T) = res(F) + max(res(T_1), res(T_2), \ldots, res(T_i)) \leq R$$

5.3.2 Predicting the Maximum Resource Usage

In order to determine whether the resource usage for executing a functionality satisfies the resource requirement of the client device, we need to be able to predict the resource usage of a facet execution tree. As mentioned before, the resource usage of an execution tree can be calculated by:

$$res(T) = res(F) + max(res(T_1), res(T_2), \ldots, res(T_i))$$

where i is the number of subtrees of T. This is a recursive formula, which in turn requires us to calculate the resource usage of its subtrees. That is, calculating the resource usage of a tree eventually requires the resource usages of all the facets in the execution tree to be calculated. The problem now reduces to finding the resource usage of a facet.

The resource usage of a facet can be divided into two parts: the static resource and the dynamic resource. The static resource usage depends on the code size, and can be known at compile-time. On the other hand, finding the dynamic resource usage of a facet is not easy, since it depends on the execution that is dynamic. The resource usage may also depends on the input, which cannot be known until run-time. Being unable to determine the actual resource usage of a facet, a better approach might be to predict its maximum resource usage. This prediction seems to give a good prediction, because the maximum resource usage gives an upper bound, and the facet can be safely executed and completed in the client device if the maximum resource usage can satisfy the resource requirement.

However, determining a maximum resource usage of a facet is also not easy. It is difficult to determine an absolute maximum because we do not know under what situation (or input condition) will this occur. The maximum found by a number of executions might only be a local maximum, but not an absolute maximum. This implies that a pre-defined (local) maximum is needed. This maximum resource usage is set by the facet developer, such that under most circumstances, the resource usage will not exceed this threshold. But, the problem is, there seems to be no fixed guideline on how to determine this threshold. If this threshold is set to be too high, taking the maximum of all the subtrees when calculating the resource usage of a functionality results in a

very large value, which is not likely to satisfy the resource requirement of the client device. If the threshold is set too low, the predicted resource usage of the functionality is likely to satisfy the client device, but the actual resource usage may not. This results in the service not being able to be completed in the client device.

Therefore, in order to have a reasonable prediction of a facet's maximum dynamic resource usage, it is better if an approximation of the maximum input size to be given at run-time is known. Knowing this approximation helps to predict a maximum dynamic resource usage of the facet that would be used. But, different user, or even different executions of the facet from the same user, require a different maximum input size. Some of them needs a smaller maximum, while others might need a larger one. It is difficult to have an approximation of the maximum input size that suits all of them, which makes it difficult for the facet developer to specify a value for the maximum dynamic resource usage of a facet. Instead of being a fixed number, this value needs to reflect the maximum dynamic resource usage of the facet for different input sizes. This is best suited by a formula that indicates the dynamic resource usage of a facet. In this case, the maximum dynamic resource usage does not need to be pre-determined, but can be adjusted at run-time for different input sizes under different situations.

With this idea of predicting the maximum dynamic resource usage of a facet, let us see how this helps in the prediction of the resource usage for executing a functionality.

5.3.3 Predicting the Worst-Case

In order to have a reasonable prediction of the resource usage for executing a functionality without aiming at an accurate facet execution tree, we use a ***worst-case*** approach. By worst-case, we refer to the situation that the client would use the most resources for executing a functionality. If, under this situation, the predicted resource usage for executing the functionality can still satisfy the resource requirement of the client device, it should be able to do so at run-time.

The idea is simple, as the worst-case approach effectively provides an upper bound of the resource usage for executing a functionality. But, it requires a definition of the worst-case. As mentioned, worst-case refers to the situation that the most resources would be used for execution; and the resource usage for executing a functionality can depend on the user input. So, if the user input that would be used in the worst case can be known by the proxy system, then the worst-case resource usage for executing the functionality can be calculated. This worst-case input (i.e. the largest input size that would be used) can be estimated by the user, and made known to the proxy system at the

time of request.

Although the idea is simple, it requires reasonable arguments to be suitable for functionality adaptation. These arguments can be seen later on in this chapter, when the safeness of the prediction is considered.

In order to calculate the worst-case resource usage, the proxy system requires a few assumptions:

- **The dynamic resource usage of a facet is represented by a formula.** As mentioned before, the resource usage of a facet is not static, and varies at run-time, probably with different inputs. For example, consider a facet for matrix multiplication that can compute matrices of different sizes. An input of two 2×2 matrices requires less memory to be allocated for the calculation, and thus less resource usage, than an input of two 10×10 matrices. Therefore, it is impossible to pre-determine the resource usage of a facet without knowing the information of the input. Even if a "maximum" resource usage is specified, so that the actual resource usage does not exceed the specified value for any inputs that would be used, it is difficult to have a pre-determined value that is suitable for all situations. This makes it impossible to have a fixed value for the resource usage of a facet. But rather, the resource usage is better represented by a ***resource formula***, so that different resource usages can be estimated for different inputs. This allows a suitable "maximum" resource usage to be estimated for a user's input. This formula is a function of the input size, i.e. $f(input\_size)$, so that the resource usages for different inputs can be estimated by the formula. For example, the dynamic resource usage of the matrix multiplication facet may be described by the formula $f(m_1, n_1, m_2, n_2) = 3(m_1 n_1 + m_2 n_2) + 2(m_1 n_2) + 10$, where m_1, n_1, m_2, n_2 are the dimensions of the two matrices. In order to allow individual facets to be easily developed without worrying about the resource usages of other facets that are required for execution at run-time, the formula only represents the resource usage of a single facet, i.e. a local resource usage.

- **The dynamic resource usage of a facet increases with the input size.** Besides having a formula that allows different resource usages to be estimated for different input sizes, we also require the resource usage to increase with the input size. The resource usage formula, in this case, is, thus, a ***monotonic increasing*** function. This requirement of the resource usage is needed for the worst-case approach so that the worst-case input (i.e. the largest input size that would be used) reflects the "maximum" amount of resources to be used. This gives an upper bound of the resource usage of a facet, and makes it easier to estimate the resource usage for

executing a functionality. This assumption is quite practical in the sense that a larger input size usually requires more memory to be allocated, and thus a larger resource usage.

- **The capability of a facet is specified by a range of inputs.** Facets usually cannot be assured to work for all inputs. They are just assured to work for a range of inputs. This range of inputs specifies the capability of the facet, which is the ability of the facet for achieving the specified functionality. We call this range of inputs the ***capability range***. The facet is assured to work as desired if the input is within the capability range. Otherwise, the functionality cannot be guaranteed. This capability range is, thus, considered part of the contract, and can be specified as the pre-conditions, i.e. the conditions that are required to be satisfied in order to have the desired functionality assured. For example, a matrix multiplication facet may only work for matrices whose dimensions are not greater than 100×100. The capability range can, therefore, be represented by $m \leq 100, n \leq 100$ where m and n are the dimensions of the matrices. The capability range, in practice, can be the maximum input sizes that have been tested and achieved the desired functionality.

- **Each request is accompanied by an approximate input size.** In order to allow the dynamic resource usage of a facet to be estimated properly, an approximation of the input size is needed. This approximation can be a range, which specifies the range of input that would be used. This range, called the ***requesting range*** is different from the capability range that a facet specifies for its functionality to be assured to achieve. Instead, the requesting range is concerned about the sizes of inputs that would be used for execution. For example, when a multiplication of two matrices is required, a user might only compute matrices of size of at most 50×50, instead of the largest size (e.g. 100×100) that a facet is capable to compute. In this case, the user can specify a requesting range of $m \leq 50, n \leq 50$ where m and n are, as usual, the dimensions of the matrices. This requesting range is accompanied with each request, as it should be known at the time when the functionality is needed. At the time of request, the requester should have a knowledge, or at least a brief idea, about the target device that would be used for executing the requested functionality, and thus an approximate range of input that would be used.

With these assumptions in mind, a detailed mechanism of the worst-case prediction can then be given. As mentioned before, it is difficult to predict an accurate facet execution tree, and hence the resource usage, for executing a functionality. Therefore, instead of predicting such an execution

tree, the proxy system tries to predict a number of possible facet execution trees that might be taken for executing the functionality. These facet execution trees are predicted based on dependencies that are needed for fulfilling the functionality. These dependencies are written as requests descriptions in the service codes and are reflected in the facet shadows. Each of these requests descriptions include a description of the functionality, as well as an approximate requesting range of the inputs that would be used for execution. Facets that match these descriptions (i.e. passed the functionality filter) are candidates to be used at run-time, and the possible facet execution trees are formed from these combinations. Note that the resource requirement is not yet taken care of by these execution trees at the current moment.

From these facet execution trees that are possible to be taken at run-time, the proxy system then selects the one(s) that might be used in the worst case. The worst case for each request can be known from the requesting range. The requesting range specifies an approximate range of inputs that would be used for execution. The largest input, thus, represents the worst case, as the largest amount of resources is used. Therefore, if the worst-case input of each request is used for calculating the resource usage of a corresponding facet, the resulting resource usage of the facet execution tree is the resource usage that would be used by the execution tree in the worst case.

However, not all of these worst-case execution trees can be used by the client at run-time. They need to satisfy the resource requirement in order to be used; i.e.

$$res_{worst}(T) = res_{worst}(F) + max(res_{worst}(T_1), res_{worst}(T_2), \ldots, res_{worst}(T_i)) \leq R$$

In general, there can be more than one facet execution tree that can be used by the client in providing a functionality using the worst-case inputs. Therefore, we need a way to determine which of these facet execution trees are better for our prediction. As one may have noticed that these possible execution trees can be used for achieving the same functionality specified by the client, the decision can be seen as selecting an execution tree that is better for executing a functionality. Our view on this is that, if all of these facet execution trees can provide the functionality in the worst case, it might be better to select the one that uses the least resources. So, base on this idea, we select the one that uses the least resources in the worst case; and the resource usage of this facet execution tree is the predicted resource usage for executing the functionality. That is,

$$res_{worst}(T) = min(res_{worst}(T_A), res_{worst}(T_B), \ldots)$$

where $T_A, T_B, \ldots$ are the facet execution trees that can be used to provide the functionality in the worst case.

As most of the decisions made in the prediction is based on the resource usage for executing the functionality provided by a facet when the worst-case input is given, it is better to have a complete picture of how the resource usages in the worst case can be predicted. Now, let us assume that there are three facets F_A, F_B and F_C to be processed by the resource filter. In order to filter facets whose functionality can satisfy the resource requirement of the client device, the proxy system has to calculate the resource usage predicted to be used by these facets and their dependencies for providing the functionality. These resource usages are represented by $res(T_{F_A}), res(T_{F_B})$ and $res(T_{F_C})$, which are, in fact, the resource usages of the execution trees rooted at facets F_A, F_B and F_C respectively.

Consider one of these facets, say F_A. The resource usage for executing the functionality provided by F_A depends on its dependencies. If different facets are used for these dependencies are used, the resource usages for the execution would be different. The number of dependencies a facet requires depends on its implementation. A facet can perform the functionality all by its own (i.e. no dependencies), or it can have any number (> 1) of dependencies. Borrowing the concepts of regular expressions, a facet can be represented by:

$$\texttt{facet} \rightarrow \texttt{(functionality)}*$$

which means a facet is depends on *zero or more* dependencies. Base on this expression, the resource usage of a facet execution tree is, thus, dependent on the resource usage for executing the functionalities (which are the dependencies of the facet). Moreover, as we have mentioned in Chapter 3, client devices in our model have the capabilities of discarding facets after use. Therefore, the dependencies of a facet are not able to use the resources in the client device at the same time. They are used one after another. Only the one that uses the most resources matters, and the resource usage for executing the functionality provided by F_A can be calculated by:

$$res(T_{F_A}) = res(F_A) + max(res(functionality_1), res(functionality_2), \ldots)$$

where $functionality_1, functionality_2, \ldots$ are the functionalities required by F_A. If F_A does not require any dependencies to help fulfilling its functionality, the last part of the equation is 0, and the resource usage becomes the resource usage of the facet itself; i.e. $res(T_{F_A}) = res(F_A)$.

The above formula requires the proxy system to calculate the resource usage for executing the functionalities $functionality_1, functionality_2, \ldots$, etc. In order to calculate this resource usage, we need to know what a functionality depends on. Functionalities are provided by facets, and there can be more than one facet implementing the same functionality. In general, a functionality can be implemented by any number of facets (including zero). This idea can also be represented as a regular expression:

$$\texttt{functionality} \rightarrow \texttt{(facet)}*$$

This implies that the functionality is depends on the facets implementing it, and the resource usage for executing a functionality, thus, depends on the resource usages of the facets that can provide the functionality. As all of these facets can be used for providing the same functionality, the one that uses the least resources is considered to be the resource usage for executing the functionality. That is, the resource usage for executing a functionality can be calculated by:

$$res(functionality_i) = min(res(T_{F_{i1}}), res(T_{F_{i2}}), \ldots)$$

where $F_{i1}, F_{i2}, \ldots$ are facets that can provide the functionality $functionality_i$. In case where there are no facets that implement the functionality, the resource usage for executing the functionality is set to be a very large value, such as ∞, so that any facet execution tree that requires this functionality are considered to use a large amount of resources and cannot be selected.

The facets $F_{i1}, F_{i2}, \ldots$ can, in turn, require other dependencies for fulfilling their functionalities. The calculation of the resource usages of these facets in providing the functionalities is the same as calculating the resource usage for executing the functionality provided by facet F_A. Since the resource usage for executing the functionality provided by a facet, $res(T_{facet})$, eventually requires calculating the dynamic resource usages of the facets involved in the facet execution tree, and the resource usages of these facets are base on the worst-case inputs, $res(T_{facet})$ is, in fact, the resource usage predicted to be used in the worst case. Also, when there are more than one facet execution trees whose resource usage in the worst case can satisfy the client device, the one that uses the least resources is selected. Therefore, the predicted resource usage, $res(T_{facet})$, is actually the *minimum* resource usage that is predicted to be used in the worst case. Figure 5-5 shows an example.

After calculating the predicted resource usages of the facets F_A, F_B and F_C, the proxy system can then compare them, one by one, with the resource requirement of the client device. Facets whose predicted resource usages satisfy the resource requirement are then filtered. The satisfiability

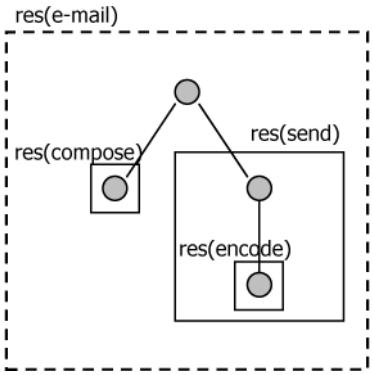


Figure 5-5: An example showing how the dynamic resource usage can be calculated by using a worst-case prediction.

of these facets can be processed by the following predicate:

$$res(T_{F_i}) \leq R$$

where F_i are the facets that are to be processed by the resource filter; i.e. F_A, F_B and F_C in this case. A facet is filtered if a "true" results in the evaluation of the predicate.

5.4 Analysis of the Conservative Prediction

With the conservative prediction described in the last section, this section tries to tackle the second question: is the conservative prediction a good prediction? Since the main concern of our proxy system is to adapt facets to the client, the facets that are returned to the client need to be suitable for both the client device and the device user, according to the execution context. In order for a returned facet to be suitable for the client device, it needs to be able to execute and complete its functionality in the device. On the other hand, the returned facet is suitable for the device user if it satisfies the preferences and other contexts expected by the user. Therefore, if our prediction does not hinder the decision of selecting a suitable facet according to the execution context at run-time, and allows all facets returned to the client to be executed and completed in the client device, then the prediction is considered to be good.

5.4.1 Safeness

Before determining whether the conservative prediction is ***safe***, we need to, first, define the safeness of a prediction. For a prediction to be considered safe, it should always lead to facets being returned able to execute and complete in the client device. It cannot result in any situation that the facets returned are unsafe for execution (i.e. the execution cannot be completed). In other words, once a facet is returned to the client, it guarantees there must exist a facet execution tree for the service to be provided in the client device.

In resource filtering, facets to be filtered are determined by the predicted resource usages for a facet to execute the functionality; and these predicted resource usages are all calculated based on the largest input sizes that would be used for execution. These largest input sizes can be known from the requesting ranges provided by the requesters of the functionalities. They are normally the largest values of the requesting ranges, such that no inputs larger than these values would be used for the executions. Furthermore, the dynamic resource usages of the facets are assumed to increase with the input sizes. That is, the larger the input size, the larger is the dynamic resource usage of a facet. By using the largest input sizes for calculating the dynamic resource usages and *assuming there is only one facet execution tree for each facet*, the proxy system is able to know the largest resource usage that would be used by the client in executing the functionality and make suitable decisions regarding facets to be filtered. This calculated value can be used as the predicted resource usage for executing the functionality, which is the largest amount of resources that the client is expected to use for execution. Safe execution can be assured because the information about the largest input sizes are provided by the requesters, who are going to use the requested functionalities and should be able to provide an approximate range of input sizes to be used for the execution. At run-time, the actual input sizes used in the executions fall within the requesting ranges, making the resource usage for executing the functionality cannot exceed the predicted resource usage, which is the resource usage predicted for the worst case. As the facets returned must have the predicted resource usages satisfying the resource requirement of the client device, and the amount of resources used in executing the returned facets cannot be greater than the predicted resource usages, the requested functionalities must be able to execute and complete in the client device. The prediction is, therefore, considered safe if the facet execution tree being predicted is used for execution at run-time. Figure 5-6 shows that the prediction is safe.

For more practical situations, during the prediction process, there are usually more than one

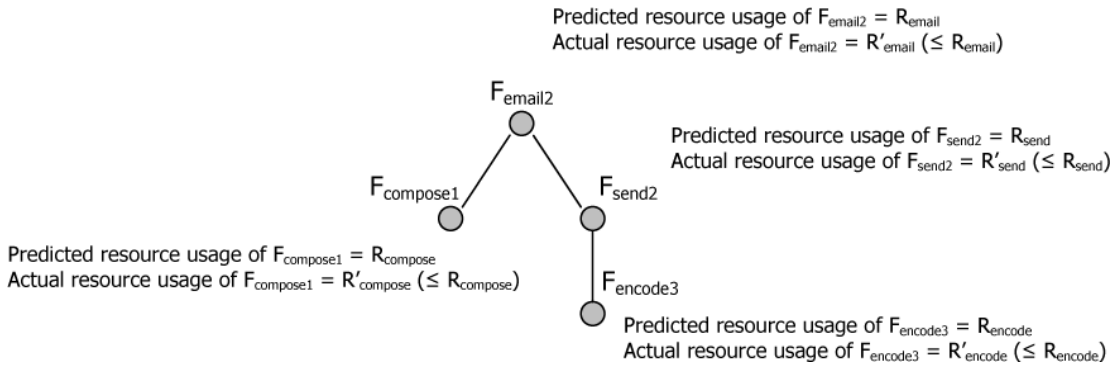


Figure 5-6: Worst-case input size guarantees safe prediction.

facet that can provide the desired functionality of each dependency (i.e. there are more than one facet execution tree for a functionality). Figure 5-1 in section 5.1 gives an example. In that case, for each of the functionality, the one that uses the minimum resources for providing the functionality in the worst case is selected. The facet execution tree predicted is, therefore, consisting of facets that can provide the functionality in the worst case using minimum resources; and the predicted resource usage of this facet execution tree is the minimum resource usage for executing the functionality in the worst case. This means that the facets returned are predicted to have their functionalities completed in the client device with the minimum resource usages. Although minimum resource usages are being considered, it still does not violate the safeness of the prediction. This is because the worst-case execution is being considered in getting this minimum resource usage for executing a functionality, where the resource usage for the execution is expected to be the largest. Even though the facets selected to be used at run-time for fulfilling the functionality may be different from the predicted ones, their resource usages for executing the functionalities must satisfy the run-time resource requirement of the device because they have to go through the resource filtering process before being selected and returned to the client. Therefore, the prediction that a facet can provide the functionality in the device cannot cause the actual execution to be unsafe, due to the resource-filtering conducted at run-time before a facet is actually returned. As long as there is a facet execution tree that is able to complete the execution for the worst case in the client device (independent of the amount of resources predicted to be used), the facet being returned can complete its execution safely. In the worst case, this execution tree to be used at run-time is only the same as the predicted one, but cannot cause the execution unable to complete in the client device. The facet returned for the execution of the required service suggests that there is at least a facet execution

tree being predicted and whose functionality can be completed in the client device. This implies the prediction is safe.

5.4.2 Adaptivity

Adaptivity is the ability of the prediction to be adapted to the execution context. If the prediction is fixed and cannot be changed according to the execution context, then the proxy system is able to predetermine the facet execution tree that would be used for the execution. No run-time information is then needed for prediction, and the proxy system is not really intelligent. On the other hand, being able to adapt allows the prediction to better reflect the decisions made for the execution context at each request, and therefore closer to the actual selection decisions at run-time.

The conservative prediction is not only safe, but also able to be adapted to the execution context. During the prediction, its purpose is to predict the amount of resources that would be used for fulfilling the functionality. It only aims at providing a guarantee for the client that the facet returned is able to complete the functionality in the client device. There is no guarantee that the facets to be used in the near future for fulfilling the dependencies have to be the ones being predicted in the facet execution tree. In other words, the facet execution tree predicted is just for predicting the resource usage for the functionality. At run-time, when more information about the execution context is known, the proxy system is able to make better predictions, and the prediction of the facet execution tree can be refined.

Let us take a look at an example to see how the prediction can be refined, with more information about the execution context available at run-time. Assume that at time t_1, the client requests a facet for executing a certain functionality, say $functionality_a$; and the device has 500KB memory available for the execution. The proxy system, after making the decision, returns a facet F_a that is suitable for the client according to the execution context at time t_1. Besides returning a facet to the client, the proxy system has also predicted a facet execution tree T, rooted at F_a, that is able to provide the functionality for the client. This facet execution tree is being predicted using the resource usage of 500KB, i.e. the execution context at time t_1. During the execution of F_a, the facet requires another functionality, say $functionality_b$, to help fulfilling its own functionality. The client, therefore, sends a request to the proxy system, at time t_2, asking for such a facet. Together with the request, the client also needs to send the new resource requirement of the client device to the proxy system. The resource requirement, at time t_2, is no longer 500KB because the execution of facet F_a has used up some of it. Assume that F_a has used 50KB of memory during its execution.

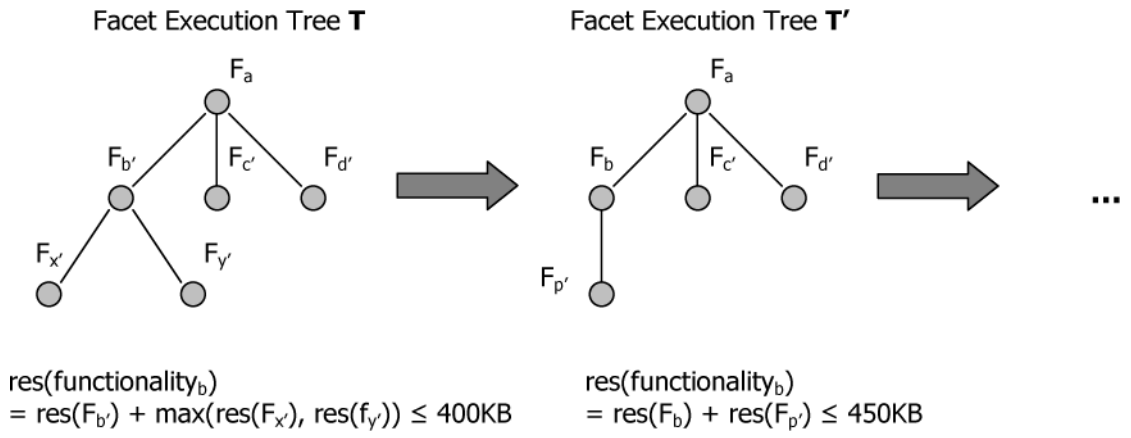


Figure 5-7: An example showing the refining of the prediction. In the execution tree T', facets F_a and F_b are known by run-time information, whereas the others are predicted with the context information at t_2.

The amount of resources left for the remaining execution is, thus, 450KB. Upon receiving the new resource requirement of 450KB, the proxy system tries to select a facet that is suitable for the new context, instead of simply following the facet execution tree T predicted at time t_1 and return the predicted facet. The facet selected is not likely to be the same as the one predicted in the execution tree T because during the prediction at time t_1, the subtree predicted for the functionality $functionality_b$ is for a resource usage of less than 450KB. At time t_1, the resource usage for the facet F_a is not known, and is predicted to be a larger value than the actual resource usage of 50KB (due to the worst-case prediction), resulting in a smaller value for the remaining execution. On the other hand, at time t_2, the facet execution tree predicted for the functionality $functionality_b$ is taking into account the execution context at that moment (which is the resource requirement in this case). Due to the difference in execution contexts, the facet execution tree predicted at time t_2 is not likely to be the same as the corresponding subtree in T. The facet execution tree T is, therefore, refined to the new facet execution tree T' predicted at time t_2, which is more suited to the new execution context. Figure 5-7 shows the refinement process.

Although the facet execution tree being predicted is refined, it does not affect the safeness of the prediction, or the facets that have been returned to the client for execution, as can be seen from the last subsection. This is because the resource filtering process has to be gone through during the refinement and the selection process, and there is at least one facet execution tree that has been predicted for the previous execution context guaranteeing the facet returned is safe for execution.

With more and more run-time information available, the prediction can be refined to be better

suited to the client according to the execution context. To see why the refinement is better when more and more run-time information is provided, let us take a closer look at the refinement process. First of all, let us assume that the proxy system has selected a facet F_x for executing the functionality $functionality_x$ specified by the client. Together with the facet returned, it has also predicted a facet execution tree T for the client's resource requirement of R at time t_1. Also, assume the local resource usage predicted to be used by the facet F_x is R_x in the worst case. At run-time, the facet F_x is executed for achieving the required functionality. However, the amount of resources it uses is only $R'_x(< R_x)$. Since it has not used up all the resources predicted to be used for the facet, more resources, $R - R'_x(> R - R_x)$, are left for the remaining execution. At time t_2, the proxy system receives a request for a facet that can provide $functionality_y$, which is required by F_x to fulfill its functionality. The proxy system tries to match a set of facets that can achieve such a functionality. This set of facets S_{run} that are matched at t_2, cannot be less than the ones being matched in the prediction $S_{predict}$ at time t_1 because the amount of client's available resources at t_2 is larger, resulting in more facet execution trees that can satisfy the new resource requirement. The set of facets being matched at run-time is, actually, a superset of the predicted one; i.e. $S_{run} \supseteq S_{predict}$. After the functionality matching, S_{run} needs to gone through the resource filter. Only those that have their functionalities being able to be executed and completed in the client device are filtered. Therefore, even though facets that are in S_{run} but not in $S_{predict}$ (i.e. $S_{run} - S_{predict}$) are not the ones predicted in T for fulfilling $functionality_y$, they still need to go through the resource filter (to satisfy the run-time resource requirement) before they can be selected. The facet being selected at run-time is, thus, safe. The set of facets that have been filtered after the resource filter is called $S'_{run}(\subseteq S_{run})$, in which a facet that best-suits the execution context at that moment (i.e. t_2), is selected. With a different facet being selected instead of the one predicted in T, the facet execution tree is refined. The new facet execution tree T' is predicted with the new resource requirement $R - R'_x$ for the functionality $functionality_y$, instead of the resource requirement of $R - R_x$ for the same functionality in T. This new facet execution tree, reflecting the execution context at t_2, is better predicted than T because the root facet for executing $functionality_y$ in T' must be the one that best suits the execution context at t_2; while the one in T is not. In the worst case, if the proxy system cannot find a better facet than the one predicted in T, it can simply return the facet predicted at t_1 and $T' = T$, but cannot be worse. Therefore, with more and more run-time information available, the proxy system is able to refine its prediction so that it is better suited to the execution context. The prediction can, thus, said to be adapted to the execution context.

5.5 Summary

In this chapter, we have covered the problem of resource filtering: how to filter facets whose functionalities satisfy the resource requirement of the client device. During the filtering process, future prediction is required. We have used a conservative approach in predicting the resource usage for executing the functionality provided by a facet, which is the minimum resource usage when executing with the largest input sizes. In the last part of the chapter, we have also showed that this prediction is safe and can be refined according to the run-time execution context.

After resource filtering, all the facets being filtered should be able to execute and complete their desired functionalities in the client device. These facets are then passed to the selection phase for selecting a facet that is considered best suited to the execution context.

With a thorough understanding of how functionality adaptation is achieved in our proxy system, the next chapter gives the design and implementation of a simple prototype to prove our concepts.

Chapter 6

Prototype Design and Implementation

With the ideas mentioned in the last few chapters, we have implemented a simple prototype for the proxy system as a proof-of-concept. The proxy system demonstrates the idea of functionality adaptation by selecting a facet for the client that provides the desired functionality satisfying the execution context and is able to execute and complete in the client device. Besides achieving functionality adaptation, the proxy system is also designed with other features that makes it suitable to be used in pervasive computing environments. These features include reasonable efficiency, user mobility support and co-operative support in making decisions for the client.

6.1 Design Issues

The aim of our proxy system is to have the intelligence of making suitable decisions on the code components that are needed for clients' executions. This feature is needed in pervasive computing environments so that code components can be downloaded anytime, anywhere, and to any device for execution. There are, in fact, a few issues that are needed when designing such a proxy system for pervasive computing environments. They include:

- **Selection Decision.** This is the main issue of the proxy system. In helping the clients to make a good decision, the proxy system has to be designed with intelligence. It has to identify the conditions that are needed to determine the suitability of a code component to the client; i.e. the conditions that helps to make a good decision. With the conditions being identified, the proxy system can then have an idea about the information needed for making the decision.

- **Architecture.** The architecture of the proxy system is also considered as an important issue.

It affects how the proxy servers in the system are placed and communicated. Different architectures have their own advantages and disadvantages. We need to identify their pros and cons in order to make a good design decision for our purpose.

- **Performance.** Performance is almost always an issue in designing a system, especially a system for pervasive computing. With the aim of computing everywhere, the response time should be reasonably fast. Users usually do not have the patience to wait for a long time for the results. If the response is too slow, they are not willing to use the system. Therefore, even if the system is not optimized for performance, it still needs to be reasonably efficient in terms of time; i.e. cannot be too slow. Techniques have to be used for improving the performance of such a system, which may include improving the computation time of the system, or improving the response time by pre-computing the results, etc.

- **Mobility.** In pervasive computing environments, users are highly mobile and are likely to be working in different locations. They may be moving with their devices, or simply move on their own and borrow any computing device that they come across on the way. Continuation of work in another location might also be needed. Despite of the move, the proxy system should be able to select suitable code components for the device/user pair in the new location. Therefore, in such a highly mobile environment, it is better if the proxy system can provide some mobility support.

6.2 Design of the Proxy System

Base on the design issues, a framework for the proxy system is designed. The proxy system is a *proxy community* where all the proxy servers are located. Requests are normally handled by individual proxy servers. But if these requests cannot be satisfied by a single proxy server, other proxy servers in the proxy system can also co-operate with each other in satisfying the requests.

Each proxy server in the system is designed such that it can handle individual requests, as well as co-operate with other proxy servers. In order to handle individual requests, description of facets (i.e. facet shadows) are needed to store locally. The proxy server can then match each request with these shadows to identify the facets that are suitable for the client. If no facets can be matched, the proxy server tries to find a suitable facet from other proxy servers. Besides selecting a facet that satisfies the current request, proxy servers are also designed to predict future requests, and to pre-fetch facets

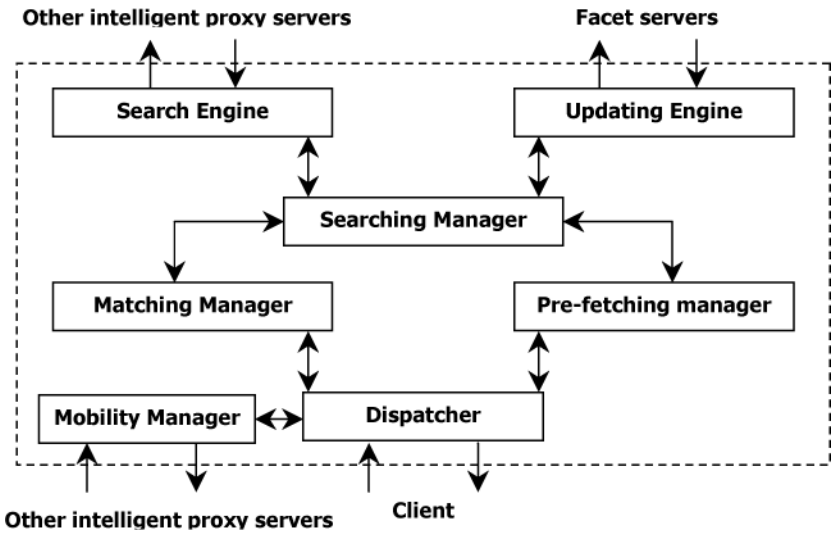


Figure 6-1: The architectural design of a proxy server.

that are expected to be used in the near future for improving the overall performance.

In our design, each proxy server has four basic managers that work closely with each other in making intelligent decisions for the clients (Figure 6-1). They are the matching manager, the searching manager, the pre-fetching manager, and the mobility manager. Besides these managers, there are also a few other components that are important to the functioning of the proxy servers. These are the main components of each proxy server, and are described below:

- **Request Handler.** The request handler acts as an interface between the clients and the proxy server. It receives requests (facet- / login- / logout-requests) from the client, and extracts the contents to be passed on to the matching manager for matching a suitable facet, or to the mobility manager for enabling mobility support. Upon receiving a suitable facet from the matching manager, the request handler returns the facet to the client, and at the same time, pass it to the pre-fetching manager for pre-fetching facets to be used.

- **Shadow Base.** This is the place for storing the facet shadows used for matching, and is the main component for the matching process. It tries to match a request with the facet shadows being stored. All facet shadows that satisfy the descriptions in the request are matched by the shadow base. They are then processed by other components for selecting the one that best-suits the client.

- **Matching Manager.** The matching manager is responsible for making intelligent decisions

for the clients. Upon receiving a request from the request handler, it passes the request to the shadow base for matching facet shadows that satisfy the request. It then selects a facet from these facet shadows, making use of the context information available for the decision. The facet being selected is returned to the client through the request handler. If, in some cases, the matching manager cannot find a facet that satisfies the request, it then passes the request to the searching manger, which in turn passes it to other proxy servers, hoping to find a suitable one with their co-operation.

- **Searching Manager.** The searching manager is responsible for retrieving facets from other proxy servers or facet servers. Whenever a facet cannot be found locally, the searching manager tries to retrieve facets from the network. It sends request to the nearby proxy servers to help finding a suitable facet for the client. Besides retrieving facets from nearby proxy servers when they cannot be found locally, facets can also be retrieved from facet servers. Facets retrieved from the facet servers are mainly for updating the cached shadow copies in the shadow base. In order to get the updates from the facet servers, the searching manager periodically sends requests to the facet servers for the updates.

- **Pre-fetching Manager.** When a facet is selected to be returned to the client, the pre-fetching manager tries to pre-fetch facets that are expected to be used by the returned facet. These facets that are expected to be used have their functionalities indicated in the shadow of the returned facet. Therefore, the pre-fetching manager makes use of these facet dependencies in the shadow for pre-fetching. These dependencies are extracted and then used as requests for suitable facets to be stored in the proxy cache for fast retrieval in the near future.

- **Mobility Manager.** Mobility support is provided by the mobility manager. Whenever it receives a logout request from the request handler, it gets all the information ready so that they can be given to other proxy servers when the client logs in in another location. We call these information the *mobility information*, which are information needed for mobility support. They include facet correlations, facets that are active in the client device when the client logged out, etc. Since the active facets are required for service continuation in the new location, the mobility manager gets these facets ready in the user's proxy cache during logout, so that they can be retrieved faster when the other proxy servers need them. During login, the mobility manager sends messages to nearby proxy servers, trying to identify the one that the client just logs out, and gets the mobility information from it. This mobility information

helps the proxy server to get the environment ready for the client, so that they are not affected greatly by the move. One example is that with the active facets information, the proxy server in the new location can continue pre-fetching facets that are expected to be used, so that the performance for the initial requests are not degraded.

6.2.1 Personalization

One of the main issue in functionality adaptation is that the facets selected have to be ***personalized***; i.e. suitable for the device user. As the preferences and usage patterns of different users are different, it is quite difficult to make decisions that suit individual users if their usage patterns are mixed. Therefore, a global proxy cache that shares among all the users cannot fit the purpose. *Personal caches* are needed to identify the usage patterns for individual users. Each user should also have a profile that stores their own preferences. All these help the proxy servers to select facets that are suitable for the users.

In order to have personal caches for individual users, the proxy server needs information about the users. User information cannot be used without the approval of the user. Therefore, *user registration* is needed so that the proxy server can get the personal information from the users that helps in selecting suitable facets for them.

6.2.2 Proxy Co-operation

Proxy servers, with local information, cannot always satisfy the user when working on their own. As these proxy servers are all handling client requests in the same manner, they should be able to get help from the others if one cannot satisfy the user. One scenario in which the proxy servers co-operate with each other is in finding a suitable facet for the client. Normally, a client request is handled locally, but the proxy server may not always have a facet suitable to be matched locally. In that case, it can ask the other proxy servers to help matching a suitable facet in their shadow bases. This co-operation, thus, increases the chance of finding a suitable facet for the client.

Besides co-operating with each other in finding a suitable facet, the proxy servers also co-operate for mobility support. Whenever a client moves from one location to another, it is likely to contact a different proxy server for handling its request. The new proxy server usually does not have much information about the client, and thus needs time to setup an environment (e.g. the personal cache) that is suitable for the client, which might result in a performance degradation during the initial requests due to cache misses. However, if the proxy server is able to setup the environment

before the client starts issuing its requests, then the performance during the initial requests can be improved. This setting up of a better environment for the new clients requires proxy co-operation.

6.2.3 Distributed Proxy Architecture

As the proxy system consists of numerous proxy servers, how the proxy servers are coupled with each other is one of the concern in designing the system. There are, in general, two kinds of architecture: centralized and distributed. Centralized architecture has all the proxy servers tightly-coupled — all the proxy servers work together and are treated as a single unit. Communications between the proxy servers are, therefore, enormous and frequent. On the other hand, in a distributed architecture, all the proxy servers are treated as individuals. They normally handle requests on their own. If, in some cases, they need to communicate with each other, they can do so but the frequency of communication is much less when compared with the centralized architecture.

Being designed for pervasive computing, proxy servers in our proxy system are better to be *distributed* so that they can handle requests independently in different locations. Having a proxy server in different locations has the advantage that requests can be sent to a nearest proxy server for reducing the access latency and improving the overall performance.

6.2.4 Context-Awareness

Proxy system designed for pervasive computing needs to be context-aware. Since clients in different contexts have different requirements, context information is needed to help the proxy system to make intelligent decisions in selecting suitable facets for the clients. These context information are mostly given by the clients, as they are usually the ones that are the most familiar with their execution contexts; e.g. available resources for execution, location, and resources nearby. Context information can be given by the clients at the time of requests, so that they can be used by the proxies for making the decisions. Apart from that, the proxy system can also keep some context information for future use. These information can be the requests of each user, which helps to determine their usage patterns. The usage pattern information can help the proxy servers to predict the user behaviors and make better decisions in pre-fetching facets to be used.

6.3 Implementation of the Proxy System

Base on the design described in the last section, a simple prototype of the proxy system has been implemented. It supports functionality adaptation by selecting a suitable facet for the client, base on its execution context, such as the amount of memory resources available for execution, user preferences, and user behaviors. Services are composed at run-time after receiving requests from the clients, and a facet that is best suited for each request is returned to the client as response.

In our prototype, Java is used for implementation, and eXtensible Markup Language (XML) is used for self-description of data and codes. Facets are implemented as Java ARchives (JAR files), each contains the shadow and the code segment of a facet. The shadow of a facet is written as an XML file, while the code segment is a package of class files with one of the classes being the main class of the facet. The main class has only one publicly callable interface and represents the only access to the functionality of the facet. Communication with the clients is done using Simple Object Access Protocol (SOAP) [47], a standard communication protocol for transferring messages written in XML format. The request handler interacting with the clients is implemented as a servlet to handle multiple requests from the clients. Requests received by the proxy servers are descriptions of functionalities and context information, written in XML format. In selecting a suitable facet for the client, the proxy system tries to match the XML request with the shadows that are also in XML format. Matching of the XML documents is based on XSet [35], an XML database and query engine from the University of California. The facet being selected is returned to the client, with the JAR file as an attachment to the SOAP message that briefly describes the facet.

Although SOAP is a standard protocol for transferring XML messages, it is not used for communication between the proxy servers. This is because SOAP is built on top of the network protocols like HTTP, which takes a longer time to generate a SOAP message for communication, and is not efficient to be used in the communication between the proxy servers. As the proxy servers are all part of the proxy system, simple communication with sockets is used.

6.3.1 Functionality Filter

During functionality adaptation, facets have to be filtered and then selected by the two-phase adaptation process. In order for the facets to satisfy the client's requirement, they have to go through two filters: functionality filter and resource filter. Functionality and resource filtering cannot be done at the same time because satisfying the resource requirement involves future prediction that is better

done at a later stage, with fewer facets.

A matching process is required for the first filter, so that facets that can provide the desired functionality can be filtered. This matching process is done in the shadow base, which contains a number of facet shadows to be matched with the request. Requests are sent as an XML query to the shadow base, and facet shadows (also in XML format) that match the requests are returned. A ***match*** here refers to the *descriptions in the request being satisfied* by any facet shadow, which is not simply an exact matching. A ***subset matching*** is used where a facet shadow is said to match the request if the request is a subset of the shadow. An XML query is a subset of an XML document if and only if every item in the query appears in the document, and the values in each of these items are exactly the same in both the query and the document. Figure 6-2 shows a few examples.

```
<facet>
<vendor>ABC Printing Co.</vendor>
<functionality>
<description>
PS to PDF conversion
</description>
</functionality>
</facet>
```

(a) Sample request

```
<facet>
<vendor>DEF Printing Co.</vendor>
<functionality>
<description>
PS to PDF conversion
</description>
</functionality>
</facet>
```

(b) Shadow that does not match the request

```
<facet>
<functionality>
<description>
PS to PDF conversion
</description>
</functionality>
</facet>
```

(c) Shadow that does not match the request

```
<facet>
<vendor>ABC Printing Co.</vendor>
<version>
<major>1</major>
<minor>b</minor>
</version>
<functionality>
<description>
PS to PDF conversion
</description>
</functionality>
</facet>
```

(d) Shadow that matches the request

Figure 6-2: Examples of a query matching the facet shadows.

The shadow base used for subset matching is modified from XSet, so that facet shadows can be returned as a result of a match. We only give a brief outline of how subset matching of the XML documents is done. For details, please refer to Ravenben's paper [35]. Facet shadows are XML documents that are merged together into a hierarchical tag index structure. The tag index is a tree

in request	not match in shadows	match in shadows
$x < 10$	$x < 5, x < 9$	$x < 11, x < 20$
$x > 10$	$x > 12, x > 15$	$x > 3, x > 8$

Table 6.1: Matching a greater capability range.

structure, with nodes being tags of the XML documents. Nested tags are represented by child nodes in the tag index. The leaves of the tag index are sets of references to facet shadows, with each set corresponding to a common value of the tag being indexed. Therefore, with this tag index, tags can be used as keys for accessing the index, resulting in references to facet shadows that match the tag value. This tag index can then be used for subset matching, where tags in the request are used for accessing the index. Facet shadows that are commonly referenced in the resulting sets are those with the request as a subset.

As we have mentioned in Chapter 4, facets that can achieve the desired functionality are not only those that have exactly the same functionality as described in the request. Facets that have the same functionality description, inputs, outputs, and post-conditions may also achieve the functionality, but with a different capability. In order to match these facets during the process of subset matching, the proxy system needs to have minor modifications of the requests. Recall that facets are said to have compatible functionality if they have a greater capability range (specified by the pre-conditions) than the one specified in the request. Matching facets of a greater capability range can be done if the proxy system is able to specify the range that needs to be matched. This range can be identified by inequalities involving "$>$" and "$<$", as in Table 6.1.

From these examples, it can be seen that if the inequality in the request involves "$<$", shadows that match a greater capability range are all those ranges with a larger upper bound. So, for a request of $x < 10$, a greater capability range is any range of the form $x <$ *any number greater than 10*. On the other hand, if "$>$" appears in the request, shadows that match a greater capability range are the ranges with a smaller lower bound; i.e. $x >$ *any number smaller than 10* for the request $x > 10$.

In order to match these capability ranges, the shadow base makes use of the ***range matching*** feature provided by XSet. This feature can match XML documents whose tag values fall within the range specified in the attribute of the tag; e.g.

<price unit="$" GE="100" LT="150" />

matches any XML documents with value between $100 (inclusive) and $150 (exclusive) in the price tag. This perfectly matches with what we would like to do in matching shadows. Shadows

Original request	Modified request
<facet> <functionality> <precondition> <variable> x </variable> <**operator**>'<'</**operator**> <**value**>**10**</**value**> </precondition> </functionality> </facet>	<facet> <functionality> <precondition> <variable> x </variable> <**operator**>'<'</**operator**> <**value GE="10"** /> </precondition> </functionality> </facet>
<facet> <functionality> <precondition> <variable> x </variable> <**operator**>'>'</**operator**> <**value**>**10**</**value**> </precondition> </functionality> </facet>	<facet> <functionality> <precondition> <variable> x </variable> <**operator**>'>'</**operator**> <**value LE="10"** /> </precondition> </functionality> </facet>

Table 6.2: Modifying the request for functionality matching.

of compatible functionality include those that have exactly the same functionality, or those with similar functionality but a greater capability range. To match all these shadows, requests have to be modified before sending to the shadow base for matching. Here is a brief idea how the modification is done. If "<" appears in the original request, then the proxy system needs to insert an attribute "GE" (greater than or equal to) with the original value as the attribute value. On the other hand, an attribute of "LE" (less than or equal to) is used if the original request involves ">". Table 6.2 shows some examples of how the requests are modified.

6.3.2 Minimum Resource Usage Tables

During the resource filtering process, the proxy servers need to predict the resource usage for executing a functionality. This resource usage prediction requires future prediction of facets that might be used. Since there are a number of possible facet execution trees (i.e. different combinations of facets) that might be used for the same functionality, the proxy system needs to make a decision about the resource usage for executing a functionality. As we have mentioned in Chapter 5, any possible facet execution tree that can be used for achieving the functionality in the worst case (i.e. with the largest input size), and its predicted resource usage can satisfy the resource requirement of the client device can be used. That means, in order for a functionality to execute and complete

in the client device, there needs to exist at least one facet execution tree that can provide the functionality in the worst case. We can, therefore, choose the one with minimum resource usage among these facet execution trees for predicting the amount of resources used in executing the functionality. Choosing the minimum resource usage for executing a functionality, say $functionality_a$, can increase the chance of another functionality (e.g. $functionality_b$), that requires $functionality_a$, to satisfy the resource requirement.

In predicting the minimum resource usage, all the resource usages for the worst-case facet execution trees need to be calculated. From chapter 5, we have the following equation for calculating the minimum resource usage of executing a functionality:

$$res(functionality_i) = min(res(T_{F_{i1}}), res(T_{F_{i2}}), \ldots)$$

where $res(T_{F_{i1}}), res(T_{F_{i2}}), \ldots$ are the resource usages for the worst-case facet execution trees rooted at $F_{i1}, F_{i2}, \ldots$, and achieving the functionality $functionality_i$. Each of these resource usages can be calculated by the following formula:

$$res(T_{F_A}) = res(F_A) + max(res(functionality_1), res(functionality_2), \ldots)$$

where $res(functionality_1), res(functionality_2), \ldots$ are the minimum resource usages for its dependencies. The resource usage of a single facet, such as $res(F_A)$, can be calculated by summing the static and dynamic resource usages of the facet. The static resource usage is a fixed value specified in the facet shadow, whereas the dynamic resource usage can be calculated by the resource formula and the largest input size that would be used. For implementation, the resource formula can be specified in the facet shadow for the proxy to parse, or a number of $data\_size/resource\_value$ pairs for table lookup. In our implementation, we use a even more simpler approach — specify it as a fixed value and modify it when needed.

It can be seen that to calculate the minimum resource usage for executing a functionality eventually implies calculating the worst-case resource usage of each of the subtrees in the possible facet execution trees. Since it is possible that the subtrees in these facet execution trees require the same functionality, resource usage calculations for these functionalities might be duplicated. It would be better if the minimum resource usage of each functionality is calculated only once. Therefore, borrowing the concept of Dynamic Programming in algorithms, each proxy server maintains a minimum resource usage table so that those resource usages that have been calculated before do not

need to be recalculated. As suggested by its name, the table should contain the minimum resource usages provided by each facet. A sample table is shown below:

Facet ID	$res_{min}(facet)$ **(in KB)**
1000345	257
1244000	531

With such a table, most of the calculations can be done statically, or in the background. The proxy servers are able to predict the resource usage for executing a functionality without the need to calculate the resource usages of its subtrees every time, if they have already been calculated. A psuedocode for calculating the resource usage of a functionality is shown in Figure 6-3.

Function: calculate-resource
Input: facet-ID, resource-table
Output: min-resource

```
IF facet-ID is in resource-table THEN
  RETURN min-resource from table
ELSE
  FOR EACH dependency of facet-ID
    facet-vector := facets matching functionality of facet-ID
    resource := find-min-resource(facet-vector)
  RETURN res(facet-ID) + maximum resource of the dependencies
```

Function: find-min-resource
Input: facet-vector
Output: min-resource

```
FOR EACH facet in facet-vector
  facet-ID := get facet ID from facet
  resource := calculate-resource(facet-ID)
RETURN the minimum resource from facet-vector
```

Figure 6-3: The psuedocode for calculating the resource usage.

However, simply maintaining such a table is not flexible enough for dynamic environments such as pervasive computing environments. In such an environment, facets are added and removed any time. There may be facets added that can help providing the functionality using less resources; or there may be facets removed that result in some functionalities unable to be achieved with the resources previously calculated. Therefore, the table needs to be updated when facets are added or removed. In order to update the table, the proxy servers need to know which entries have to

be updated. Unluckily, the table does not provide much information, and in the worst case, all the resource usages might need to be recalculated.

In spite of this, the original table is separated into two tables: ($T_{functionality}$) is the minimum resource usage table for executing a functionality, and (T_{facet}) is the minimum resource usage table for executing the functionality provided by a facet. Besides maintaining the minimum resource usages, the tables also maintain the facet that is chosen to provide the minimum resource usage for the functionality and a facet's dependency that requires the most resources. The two tables, $T_{functionality}$ and T_{facet}, are shown below:

$T_{functionality}$:

Functionality Description	$res_{min}(func)$ **in (KB)**	$facet_{min}$
matrix multiplication	257	1000345
matrix addition	531	1244000

T_{facet} :

Facet ID	$res_{min}(facet)$ **in (KB)**	$dependency_{max}$
1000345	123	matrix addition
1244000	456	–

By maintaining $facet_{min}$, the proxy system is able to know that the corresponding $res_{min}(func)$ needs to be updated when $facet_{min}$ is changed or being removed. On the other hand, when a facet is added, functionality matching is required to identify which of the entries in $T_{functionality}$ needs to be updated. In general, the entry needs to be updated if the facet added has a greater capability range than the corresponding functionality in the table entry. After identifying these entries, the resource usages of these functionalities need to be recalculated and the corresponding $res_{min}(func)$ updated accordingly.

In fact, the two tables correspond to the two resource equations given earlier. As the two equations are mutually dependent, the two tables are also dependent on each other. Whenever an entry in one table is updated, some entries in another table might be affected. Therefore, during the updating of either table, the other table has to be checked for updates as well. The psuedocodes of the updating of the tables are shown in Figure 6-4.

6.3.3 The Scoring System

With the facets satisfying the client's requirements, the one that is decided to best-suit the device user is selected. A scoring system is needed to help keeping track of the suitability of the facets

Function: add-facet
Input: func-table, facet-table, facet **Output:** func-table, facet-table

```
REPEAT
  FOR EACH entry in func-table
    IF facet's func has a greater capability than entry's func THEN
      r := calculate-resource(facet)
      IF r < entry's resource THEN
        entry's minimum resource := r
        entry's minimum facet := facet
        put entry to vector v
  FOR EACH entry in facet-table
    FOR EACH update in v
      IF entry's maximum dependency == update's dependency THEN
        FOR EACH dep in entry's facet
          r := res(dep)  // get from func-table or recalculate
        max-res := maximum r of the dependencies of the entry's facet
        max-dep := dependency corresponding to max-res
        total-resource := res(entry's facet) + max-res
        if total-resource > entry's minimum resource THEN
          entry's minimum resource := total-resource
          entry's maximum dependency := max-dep
          put entry to vector v'
UNTIL v or v' is empty
```

Function: remove-facet
Input: func-table, facet-table, facet **Output:** func-table, facet-table

```
remove corresponding facet entry in facet-table
put entry in vector v
REPEAT
  FOR EACH entry in func-table
    FOR EACH facet in v
      IF entry's minimum facet = facet THEN
        r := recalculate res(entry's func)
        if r < entry's minimum resource THEN
          update entry  // similar to the first part of add-facet
          put entry to vector v'
  FOR EACH entry in facet-table
    FOR EACH update in v'
      if entry's maximum dependency == update's dependency THEN
        update the entry  // similar to the last part of add-facet
        put entry to vector v
UNTIL v or v' is empty
```

Figure 6-4: Psuedocodes for updating the tables when a facet is added or removed.

for the user. The facet that scores the most is decided to be the most suitable for the user. In our prototype, we have used a simple scoring system described below.

In this scoring system, a facet is decided to be comparatively better if they have been used before, because they have ever satisfied the client. It is even more better if the facet has been cached for the user, because it takes less time for the proxy server to return such a facet to the client. These two are the proxy preferences in making the decision. Besides that, the proxy server also considers the preferences set by the user. They should be more important than the proxy preferences because they must be better reflecting the user's desire, while the proxy preferences are only some guesses.

Due to the difference in the importance of these preferences, different scores are given to the facet when they are satisfied. Facets satisfying the user preferences are given a higher score than those satisfying the proxy preferences. A score of 1 is given to the facet if it exists in the user's history, i.e. it has been used before. If, on top of this, the facet is also cached by the user, a score of 2 is added to the facet. A score of 4 is further added if any of the user preferences is satisfied by the facet.

However, assignment of scores based only on whether the facet has satisfied the user preferences is not enough. Two facets both satisfying the user preferences might be of different importance to the user. One might be satisfying one item in the user preferences, while another might be satisfying two. Even if only one user preference item is satisfied by both facets, which of the facets is more suitable to the user also depends on what user preferences is being satisfied. Therefore, different items in the user preferences are indicated with different scores. Users have the freedom to assign scores to these items, indicating their importances to the users. The corresponding score is added to the facet if the item in the user preferences is satisfied. The facet that scores the highest is considered to be best-suited to the client. In case of a tie, one of the facets is randomly selected to be returned.

This simple scoring system just described can help selecting a suitable facet for the device user. Base on the scores assigned by the proxy server and those indicated by the user in the user preferences, facets receive different scores according to their importance (or suitability) to the user. The one that scores the highest should be the most suitable, and is selected to be returned to the client. In fact, this scoring system is a simple one for demonstrating the selection in the prototype. Better scoring systems can be used for making better decisions for the clients.

6.3.4 Minimal In-order Pre-fetching

In order to allow computing even in small mobile devices, services in our model are made up of facets, which are on-demand downloaded to the client devices for execution. If facets were only selected by the proxy servers on-demand, performance is possibly affected. It would be much better if facets to be used by the dependencies can be pre-fetched in the proxy cache so that they can be retrieved faster when requested [29].

Facets to be used by the dependencies can be predicted by the proxy servers. Dependencies to be requested are described in the facet shadows so that the proxy servers can use them for prediction. There are usually more than one facet that are needed for fulfilling the required service, making the order of pre-fetching a concern. For better performance, facets are to be pre-fetched in the order of their requests; i.e. facets to be used in the near future are pre-fetched first. Therefore, it seems that the order of pre-fetching is along the in-order traversal of the facet execution tree (as a facet execution tree is drawn such that an in-order traversal of the facets indicates the calling sequence). If two facets are to be pre-fetched everytime a facet is returned to the client, these two facets are the facet for the first dependency and the facet for the first dependency of the facet just predicted, i.e. facets F_b and F_c in Figure 6-5(a).

If a facet returned has no dependencies, then the pre-fetching needs to go up one or more levels of the facet execution tree, and pre-fetch the next dependency of a facet. Figures 6-5(b) and (c) show two examples. The prediction becomes more complicated if the client has cached some of the facets that have been used before. It is usually quite difficult to keep track of the facets that are in the client's cache, in order to pre-fetch suitable facets for the client. For example, in Figure 6-5(c), if a facet is pre-fetched for dependency d_x but the client requests for dependency d_y instead of d_x in its next request, because a suitable facet for d_x is found in the client's cache. The mis-prediction due to the client's cache causes a performance degradation because the facet can only be selected on-demand.

Due to all these difficulties in predicting the facets to be pre-fetched, a simple pre-fetching mechanism is used in our prototype. When a facet is returned to the client, a facet for each of its dependencies are pre-fetched so as to avoid the mis-prediction due to the existence of the client's cache. Besides pre-fetching for its dependencies, a facet is also pre-fetched for the first facet to be used by the first dependency. Figure 6-6 shows the facets to be pre-fetched and their order of pre-fetching, assuming the request order of the dependencies are shown in the facet shadow. We call

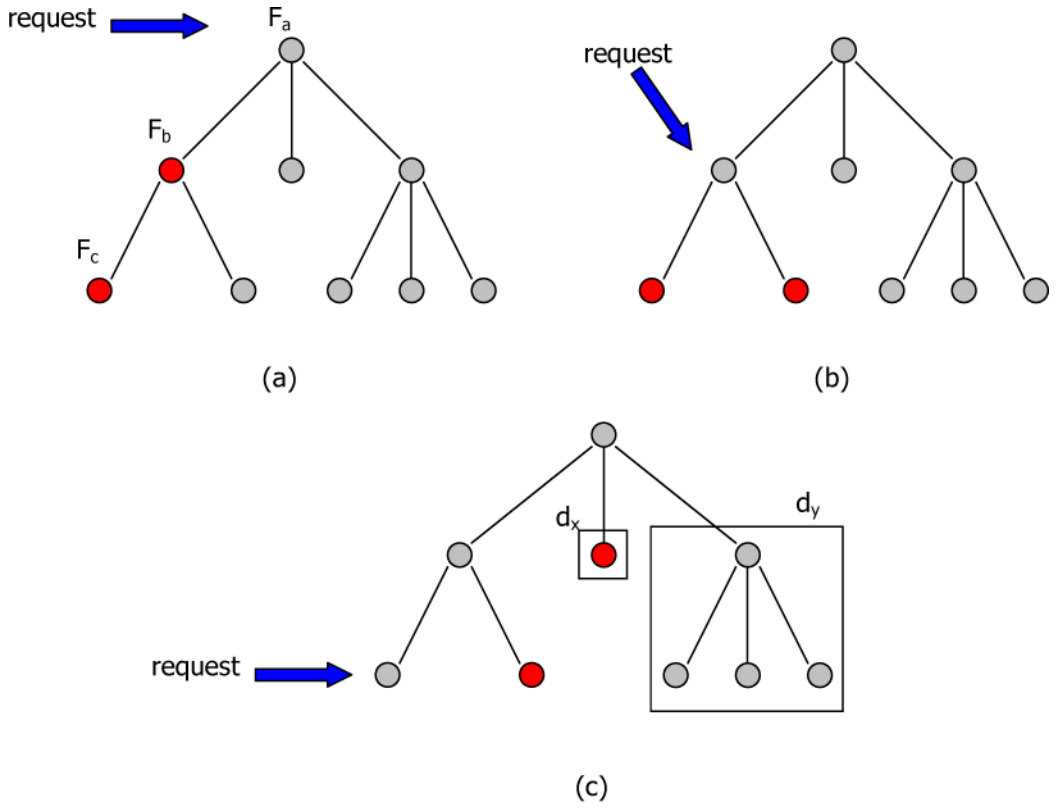


Figure 6-5: Facets to be pre-fetched in the order of their requests.

this mechanism the ***minimal in-order pre-fetching***, where the number of facets to be pre-fetched is minimal to avoid mis-prediction due to the existence of the client's cache, and are pre-fetched in the order of their requests.

Similar to the scoring system, the pre-fetching mechanism used in our prototype is also a simple one for demonstration purposes. Other pre-fetching mechanisms can be used for achieving better performance.

6.4 The Interaction Model

With the mechanisms of the proxy server described, this section gives an overall picture of the how the proxy server works and its interaction with the clients, other proxy servers and the facet servers. The working mechanism of a proxy server is shown in Figure 6-7.

When the proxy server receives a request from a client, the request is first handled by the request handler. The request is a SOAP message, containing a request description and information about its execution context (including the amount of memories available for executing the functionality and

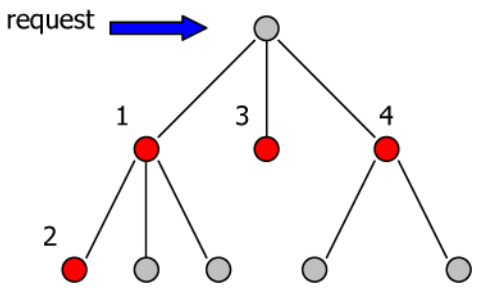


Figure 6-6: The facets to be pre-fetched in a minimal in-order pre-fetching.

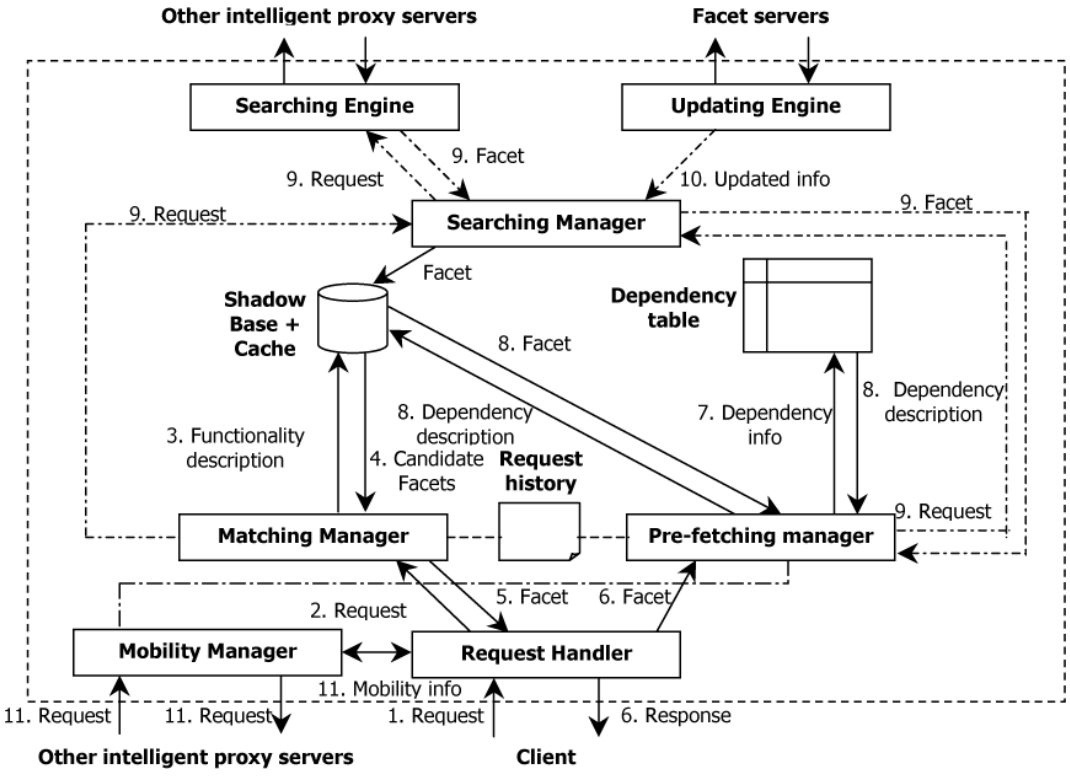


Figure 6-7: The working mechanism of a proxy server.

a user identifier) (1). The request handler analyzes the SOAP message for the type of request, and forward the message to the corresponding manager for further processing. If it is a facet request, the message is forwarded to the matching manager. In the case of a login/logout request, the message is forwarded to the mobility manager. Now, consider the case for a facet request, where the matching manager has just received the message (2). Upon receiving the SOAP message, the matching manager extracts the description and the context information from the message. The request description is then sent to the shadow base for matching facet shadows that satisfy the requested functionality (3). Facet shadows that match the functionality are returned to the matching manager for resource filtering and context selection (4). A shadow that is best suited to the client is selected during context selection, and the corresponding facet is retrieved from the cache (or the network if it is not cached). The facet is then passed back to the request handler for returning to the client (5). The facet selected is returned to the client as an attachment to the SOAP response. At the same time, the facet is passed to the pre-fetching manager for pre-fetching facets for its dependencies (6).

To pre-fetch, the pre-fetching manager extracts all the dependencies from the facet shadow, and stores them in the dependency table (7). These dependencies are then retrieved and put in a stack, in such an order that pre-fetching can be done in the requesting order. Basically, all the dependencies are put on the stack, and the first one is popped out and used for pre-fetching. During pre-fetching, the pre-fetching manager uses the same mechanisms as the matching manager (i.e. functionality and resource filtering, and context selection), except that the facet is stored in the cache instead of returning to the client. A facet is pre-fetched for the first dependency, and its dependencies are stored in the dependency table. The first dependency of this pre-fetched facet is then extracted and put on the stack, and all the dependencies in the stack is used for further pre-fetching (8).

In the actual selection (or pre-fetching), requests might not be satisfied locally by any facet shadow. In that case, the matching manager (or the pre-fetching manager) tries to retrieve a suitable facet from the network. The request description and the context information are sent to the searching manager, which contacts the nearby proxy servers for finding a suitable facet for the client. The facet found is then passed back to the corresponding manager, which returns it to the client through the request handler (9).

Besides contacting other proxy servers, each proxy server can also interact with the facet servers. Interaction with the facet servers are mainly for updating the facet shadows in its shadow base. When the searching manager receives updated information from the facet servers, shadows in the shadow base are updated accordingly (10).

For the case of receiving a login/logout request, the request handler forwards the message to the mobility manager for mobility support. In a login request, the mobility manager tries to contact nearby proxy servers and locate the proxy server that last serve the client. After locating the proxy server, a request is sent for retrieving mobility information of the client. These information are used by the mobility manager to setup the environment for the client. Furthermore, the mobility manager also contacts the pre-fetching manager for pre-fetching facets to be used by the active facets in the near future, making the environment better for the client. In the case of a logout request, mobility information is prepared. Facet correlations are analyzed by the mobility manager, and requests are sent to the pre-fetching manager for pre-fetching the active facets. These facets are pre-fetched for better performance when continuation is needed in the new location (11).

By having the intelligence of selecting a suitable facet for the client, and the ability to co-operate with other proxy servers and the facet servers, the proxy server can be used in pervasive computing environments for adapting code components to the clients.

Chapter 7

Evaluation and Discussion

With a simple prototype of the proxy system for functionality adaptation in pervasive computing environments, evaluation of the adaptation quality and performance can be done. In this chapter, experiments are conducted to test the quality of functionality adaptation and the performance achieved by our prototype. In order to have a better idea of the quality of adaptation, a metric is defined for the evaluation. Besides showing the results of the evaluation, discussions of these results are also given in the chapter.

7.1 Evaluation Platform

In our experiments, our prototype is tested to examine the quality of functionality adaptation and its performance in processing a request from a client. The proxy server is running on a PC, with the following specification:

CPU:	Intel Pentium 4 2.26GHz
Memory:	256MB
Cache size:	512KB
Operating System:	Linux RedHat 7.3 (kernel 2.4.18-3)
Softwares:	Sun's J2SE 1.3.1, Sun's Java Web Services Development Package Version 1.0_01, Apache Xerces Java Parser 1.4.4

7.2 Testing the Quality of Functionality Adaptation

Since the aim of our proxy system is to adapt service codes (facets) to the clients such that they can perform the desired functionalities on the client devices, there is a need to evaluate the quality of the functionality adaptation achieved by our proxy system.

In order to demonstrate how our proxy system adapts service codes to the clients, we have conducted an experiment to show the facets being selected by the proxy server for adapting to a client. An application is chosen and tested with several runs. A different request is used in each run so that the proxy server's ability to adapt service codes for the clients according to their run-time execution contexts can be illustrated. Base on these runs, we will then evaluate the quality of functionality adaptation achieved by our prototype.

In order to evaluate the quality, we need a way to determine whether the adaptation achieved by our proxy system is good or not. Different from content adaptation in which some existing ways (e.g. analyzing the perceptual color depth, the percentage loss in transcoding, or the bandwidth consumption) are available to evaluate the quality of adaptation, there seems to have no existing evaluation mechanisms for evaluating the quality of functionality adaptation. Therefore, we need to define our own metric for the evaluation. This metric should reflect how well the service codes are adapted to the clients. As mentioned, functionality adaptation is to adapt service codes such that they can perform the functionality needed by the client and suit the client's run-time execution context. This suggests the metric to depend on the following factors:

- **Resource Usage:** The service codes selected by the proxy server are to be executed in the client device. It would be better if the facets adapted can make full use of the resources available in the client device for execution. This resource can be the amount of memory available for execution, the available bandwidth for retrieving a facet from the proxy server, etc. Facets with larger resource usages are possibilities of providing better services to the client, and should be indicated in the metric.

- **Functionality Capability:** Facets with a larger capability but can provide the desired functionality to the client can be selected by the proxy server for functionality adaptation. Facets that have greater capabilities are more preferable than facets of relatively smaller capabilities. These facets can thus be cached in the client and have a larger possibility to be reused due to their greater capabilities. This ability of adapting a facet of a greater capability should also be reflected in the metric.

- **User Preferences:** The metric should also indicate how well the facets have satisfied the user preferences. Users requesting the functionalities in different contexts have different user preferences. The more a facet satisfies the user preferences, the better it is adapted to the client's execution context. This implies a better adaptation of the service codes for the client's execution.

Combining these factors, we have defined a metric called ***adaptation quality index (AQI)*** to calculate the quality of functionality adaptation:

$$\begin{aligned} AQI &= resource\_adapt\_index \times weight\_of\_resource + \\ & capability\_adapt\_index \times weight\_of\_capability + \\ & preferences\_adapt\_index \times weight\_of\_preferences \end{aligned}$$

In this experiment, we have assumed all the three factors affecting the adaptation quality to carry equal weights, resulting in the following adaptation quality index:

$$AQI = \frac{resource\_adapt\_index + capability\_adapt\_index + preferences\_adapt\_index}{3}$$

These weights can be adjusted according to the importance of each factor in the corresponding situation. The adaptation index is normalized to 1 for fair comparison when more factors are found to be affecting the quality of functionality adaptation later on.

The resource adaptation index ($resource\_adapt\_index$) is affected by the total resource usage of the facets returned for executing a functionality and the available resources in the client device for the corresponding execution. The closer the total resource usage is to the available resources, the better is the facets in making full use of the resources for execution; increasing the possibility of having a better service provided. This index is calculated by the following formula:

$$resource\_adapt\_index = \frac{total\_resource\_usage\_in\_executing\_a\_functionality}{available\_resources}$$

According to this definition, a larger resource adaptation index indicates a better adaptation quality.

The capability adaptation index ($capability\_adapt\_index$) depends on the capabilities of the facets returned and the largest possible capabilities among all the possible facets providing the corresponding functionalities. The closer the capability of a returned facet is to the largest possible capability, the more possible it is to be reused when cached. However, different functionalities might

have a completely different set of capability ranges. One functionality might involve capabilities of a smaller range, e.g. $size < 10$, $size < 20$, $size < 50$; the other functionality might involve capabilities of a larger range, e.g. $size < 10000$, $size < 20000$, $size < 100000$. If only the capability difference is measured, the analysis is not fair. Therefore, for a fair comparison, we use percentage difference in calculating the closeness of two capabilities:

$$capability\_adapt\_index = \frac{capability\_of\_a\_facet}{largest\_possible\_capability\_for\_the\_functionality}$$

The further the capability of a returned facet is from the largest possible capability, the better is the adaptation. Therefore, a larger capability adaptation index indicates a better adaptation. Note that there are usually more than one request involved in a single run of an application. In that case, the *average* capability adaptation index is used.

Finally, the preferences adaptation index ($preferences\_adapt\_index$) depends on the user preferences, the importance of each of these preferences, and the preferences that have been satisfied by the facet selected. This index is calculated by the following formula:

$$preferences\_adapt\_index = \frac{total\_scores\_of\_the\_preferences\_being\_satisfied}{total\_scores\_of\_all\_the\_preferences\_in\_the\_user\_profile}$$

A higher score of the preferences being satisfied implies a larger importance the facet is to the user. Therefore, a larger preferences adaptation index implies a better adaptation. Similar to the case for calculating the capability adaptation index, the *average* preferences adaptation index is used for a single run of an application.

7.2.1 Othello Application

We have tested the functionality adaptation with a chess game called Othello. This game application is designed to use facets of 19 different functionalities. Instead of only providing 19 facets for the application, 5 facets of different capabilities are designed for each functionality to allow flexibility for the proxy server in selecting a facet for the functionality. This makes a total of 95 facets available for the application. For the ease of analysis, we have assumed facets for each functionality to have the same dependencies and ignored the dynamism in this dimension. Therefore, all the different runs should request the same functionalities, making it fair for comparison and analysis. Due to the same functionalities being requested, the resulting facet execution trees for different runs should

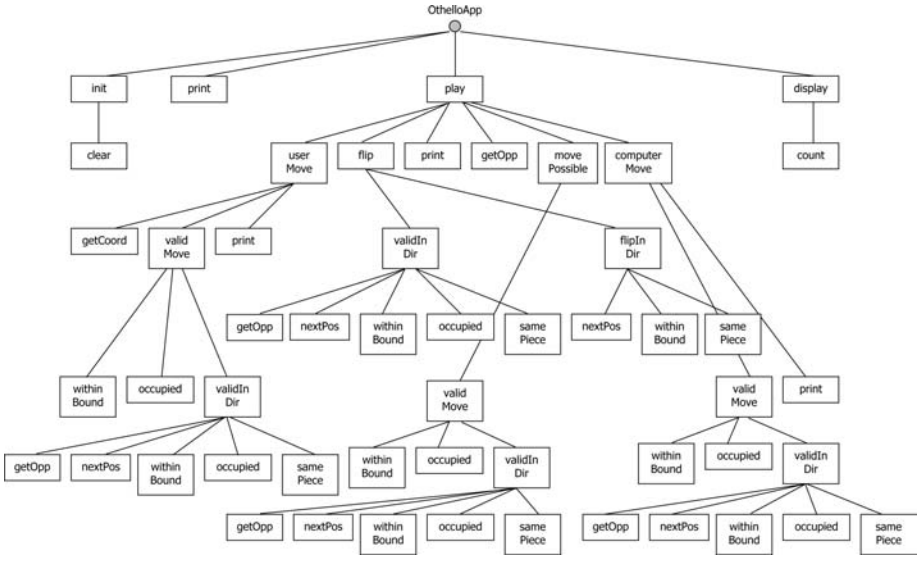


Figure 7-1: A tree showing the functionalities required by the Othello application.

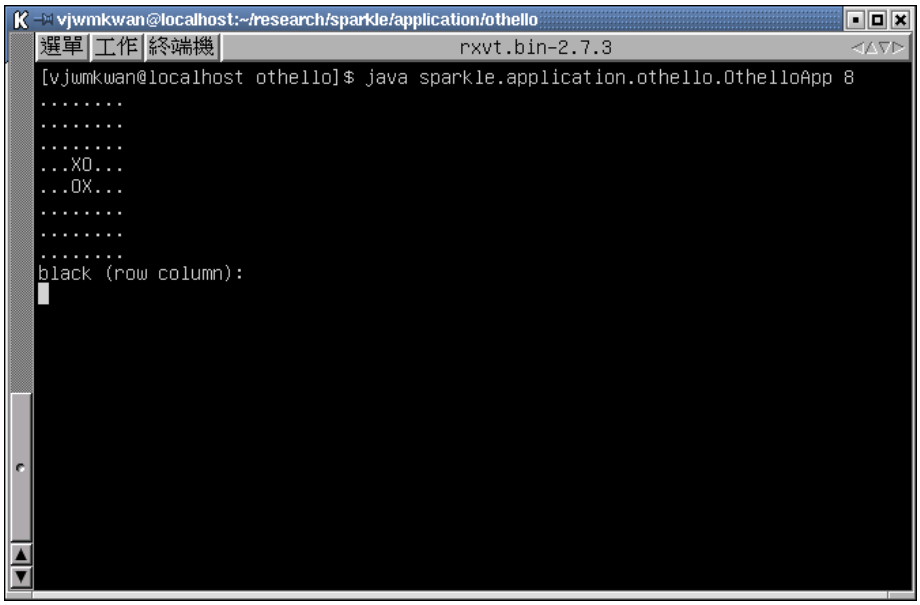
be similar, with only variations in the facets being selected. Figure 7-1 shows the functionalities required by the Othello application.

The capability of each facet is determined by the largest size of the game board that it can handle. 5 facets of different capabilities ($size < 5$, $size < 8$, $size < 10$, $size < 15$ and $size < 20$) are designed for each functionality. The proxy server needs to select a facet that can provide the desired functionality according to the requesting range.

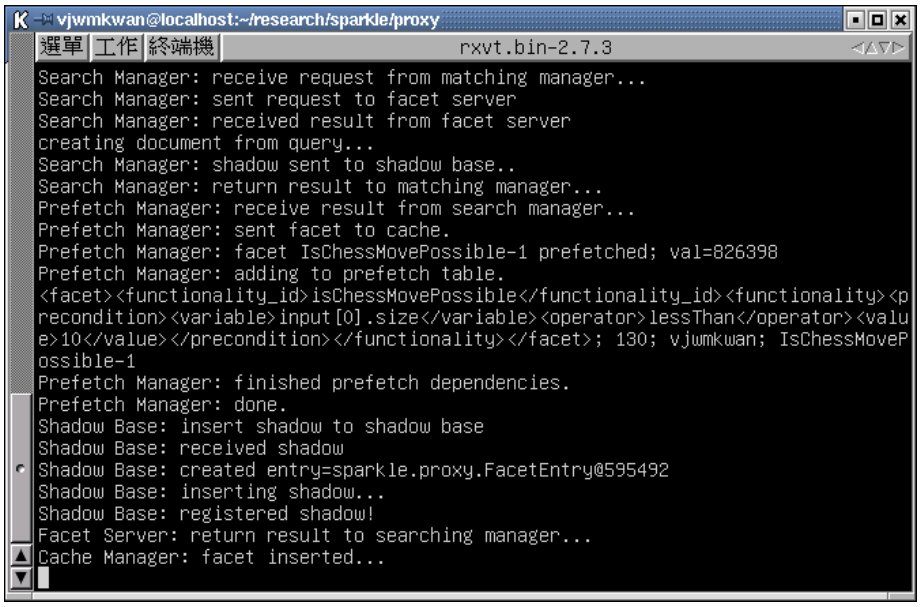
Before the start of the experiment, all the available facets are put in the facet server running on the same machine as the proxy server. They are needed by the proxy server to return a facet to the client. The proxy server is then started and facet shadows of the 95 facets are loaded into the shadow base, getting ready for processing the requests. Requests are then sent to the proxy server to examine the quality of functionality adaptation. Figure 7-2 shows a screen-shot for the execution.

The proxy server is first tested with 3 different requesting ranges, namely $size < 10$, $size < 5$ and $size < 16$. The first requesting range should allow 3 facets for each functionality to be candidates for selection, while the second and the third requesting ranges allow all or only one facet to be candidates for selection respectively. We have used the same user profile (user preferences) for these three runs. The user profile being used is shown in Figure 7-3.

The resulting facet execution trees for the three runs are shown in Figure 7-4. It can be seen that the facets being selected in these runs are, in general, quite different. They are selected according to the requesting range of the client. Since a facet is selected from all those that can provide the desired



(a) The client executing an Othello application



(b) The proxy server running to provide facets for an Othello application

Figure 7-2: Testing with an Othello application.

```
vendor:  <Paul, 6>, <Mary, 4>, <Peter, 4>
resource:  <medium(L), 2>
class:  <game, 1>
functionality_id:  <initOthello, 3>, <def, 11>
```

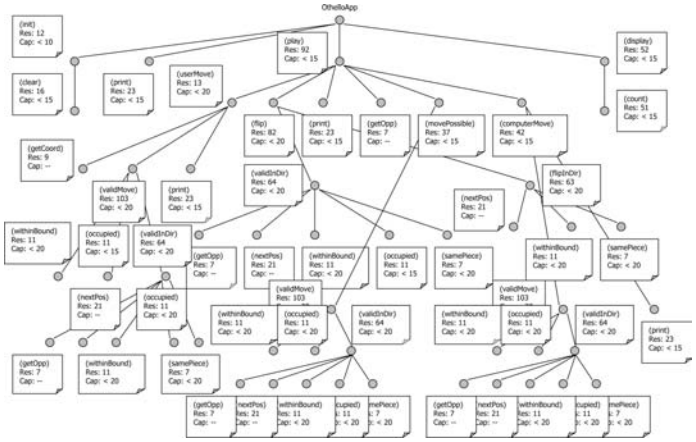
Figure 7-3: The user profile (profile 1) for examining functionality adaptation with different requesting ranges.

functionality but might be of a greater capability, more candidates are available for selection if the requesting range is smaller, thereby increases the chance of having a better facet to be adapted.

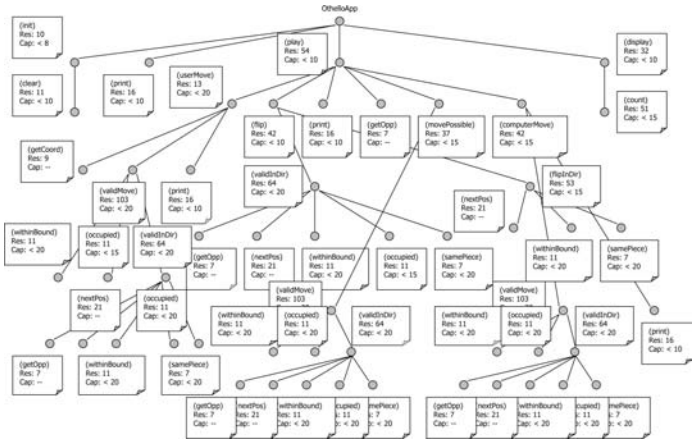
After examining the functionality adaptation with different requesting ranges, we would also like to examine it with different user preferences to see how the proxy server adapts the service codes according to these preferences. Therefore, we have tested the adaptation with a different user profile (see Figure 7-5) and examine its ability to adapt. The resulting facet execution trees are shown in Figure 7-6.

Apart from providing the facet execution trees, we also need to evaluate the quality of functionality adaptation based on these results. Using the Adaptation Quality Index (AQI) defined earlier, we should be able to evaluate the adaptation quality achieved by our prototype by analyzing the facet execution trees.

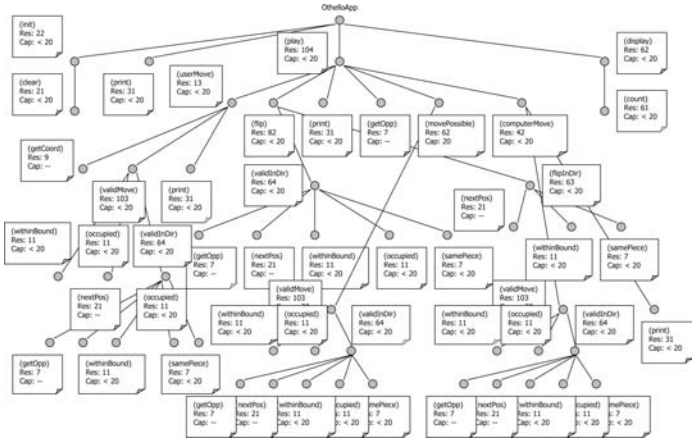
The AQIs of each run of an application is calculated and shown in Table 7.1. From the indices calculated, it can be seen that the preferences adaptation index is, in general, quite low. The unsatisfiability of the user preferences is due to the fact that there does not always exist a facet that can satisfy what the device user specifies. Therefore, this adaptation index depends largely on the facets that are available and what user preferences are specified. User preferences like “vendor”, “version”, etc. are preferences that cannot always be satisfied. The other preferences like “resources”, “performance level”, etc. can be satisfied to some extent. In this experiment, preferences such as “vendor” has been used that causes a low preferences adaptation index. Apart from the preferences adaptation index, the resource adaptation index can almost never achieve 1.00 because the facet returned is selected from the candidates that are available. These facets cannot be able to make full use of the resources available in every client devices. This is, in general, not as good as the transcoding techniques that might be able to be applied to suit each individual client; but not much can be done because transcoding technique is not suitable for code adaptation. Despite all these restrictions, the AQIs are still acceptable. It can, on average, achieve an AQI of 0.67 (in general, an AQI of more than 0.5 can be considered as acceptable). If we ignore the variations in user preferences, the aver-



(a) requesting range: $size < 10$



(b) requesting range: $size < 5$

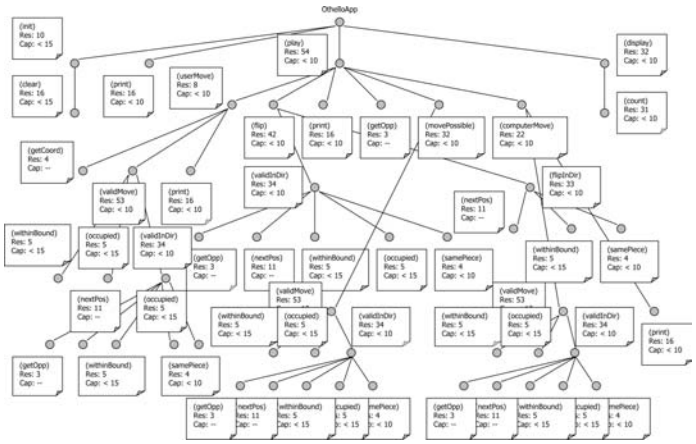


(c) requesting range: $size < 16$

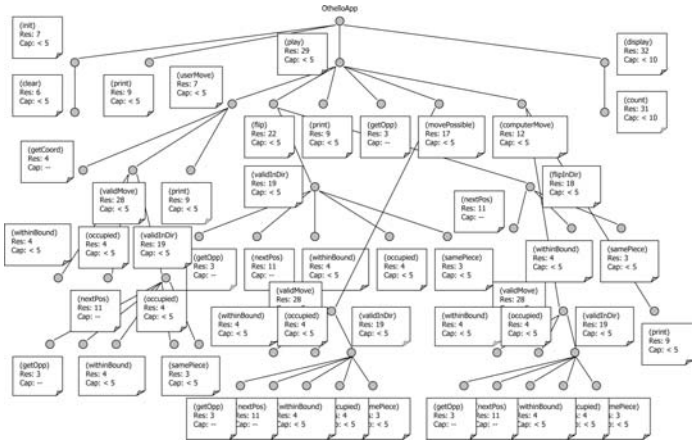
Figure 7-4: Facet execution trees for different requesting ranges.

```
vendor:  <Paul, 6>, <Mary, 4>, <Peter, 4>
resource:  <smallest(1), 4>
class:  <game, 2>
functionality_id:  <initOthello, 3>, <def, 11>
```

Figure 7-5: A different user profile (profile 2) for examining functionality adaptation.



(a) requesting range: $size < 10$



(b) requesting range: $size < 5$

Figure 7-6: Facet execution trees for different requesting ranges using a different user profile.

Run	$res.\_adapt\_index$	$cap.\_adapt\_index$	$pref.\_adapt\_index$	AQI
$size < 10$, profile 1	0.92	0.94	0.18	0.68
$size < 5$, profile 1	0.95	0.89	0.16	0.67
$size < 16$, profile 1	0.89	1.00	0.31	0.73
$size < 10$, profile 2	0.87	0.69	0.35	0.64
$size < 5$, profile 2	0.94	0.53	0.45	0.64

Table 7.1: Adaptation Quality Indices for different runs of the Othello application.

age AQI is about 0.86, which is 86% from the optimal. From these data, we can conclude that the quality of functionality adaptation achieved by our proxy system is quite good; and the quality can be enhanced if more variants of the facets with different properties are available for satisfying the different requirements of the clients.

7.3 Testing the Performance of Functionality Adaptation

Although our prototype is not optimized for performance, proxy system being designed for adaptation in pervasive computing environments should not have a poor performance. Using too much time to adapt service codes to the clients is not quite tolerable in practical situations. Therefore, there is a need to evaluate the performance of our prototype in order to have a brief idea about its suitability to be used in pervasive computing environments.

In measuring the performance of functionality adaptation achieved by our proxy system, the ***processing time*** and the ***decision time*** for returning a single facet are both measured. The processing time is the time for the proxy server to process a request; i.e. the time for the proxy server to receive a request to the time when it returns a facet to the client. In our prototype, this includes the time for the proxy system to analyze the SOAP message received from the client and to create a SOAP response for sending back to the client. The decision time is the time for the proxy server to make a decision on the facet to be returned; i.e. the time for the matching manager to receive a request to the time it determines the facet to be returned.

The Othello application is again used for the testing. The application is run for a few times (with the same requests) and the proxy server measures its processing times and decision times in handling the requests. Taking the average of all these runs gives the average processing time and the average decision time for the proxy server to adapt a single facet for a client.

Besides measuring the processing and decision times for a shadow base of 95 facet shadows, the

size of the shadow base is also gradually increased to examine the performance of the functionality adaptation with a bigger shadow base. This can also evaluate the scalability of the proxy server to be used in pervasive computing for adapting service codes to the clients. All these measurements give us a brief idea about the performance of our proxy server in functionality adaptation.

7.3.1 Performance without facet pre-fetching

In order to reflect the average performance, the requesting range sent by the client causes about half of the facets satisfying the functionality to be matched for selection. Therefore, with the 5 different capabilities of the facets mentioned in the last section, we have chosen to use $size < 10$ as the requesting range. This makes facets of capability ranges $size < 10$, $size < 15$ and $size < 20$ to match the functionality of the client. That means 3 out of 5 facets are needed to be further processed by the second phase of the two-phase adaptation in order to determine the facet to be returned.

The processing times and the decision times for performing the functionality adaptation with several runs of the application are then measured. The size of the shadow base is also increased and tested again. In this experiment, we have assumed no facets are pre-fetched in the proxy cache (by disabling any facet pre-fetching mechanism). The processing times and the decision times versus different sizes of the shadow base are shown in the graph in Figure 7-7.

7.3.2 Performance with facet pre-fetching

The same experiment is conducted with facet pre-fetching enabled in the proxy server. The performance is expect to be better when compared to disabling facet pre-fetching in the proxy server. The results are shown in Figure 7-8.

7.3.3 Overall Performance

From the graphs in the last two subsections, it can be seen that both the processing times and the decision times increases with the size of the shadow base. This is expected because a larger shadow base has a higher probability of having more facets for providing a certain functionality (i.e. more candidate facets), and the proxy server requires more time to select a facet from these candidate facets. In addition, the difference between the processing and decision times is almost constant, except some minor discrepancies when facet pre-fetching is enabled. This constant difference is due to the time for creating the SOAP message to respond to the clients. Although both the processing

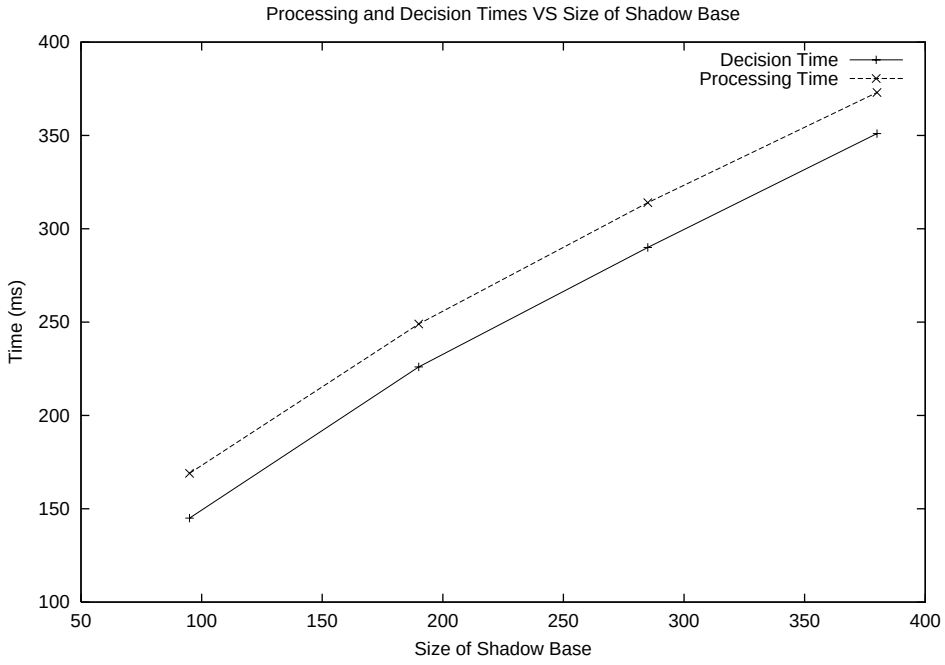


Figure 7-7: A graph showing the processing times and the decision times with respect to different sizes of the shadow base, for the case without facet pre-fetching.

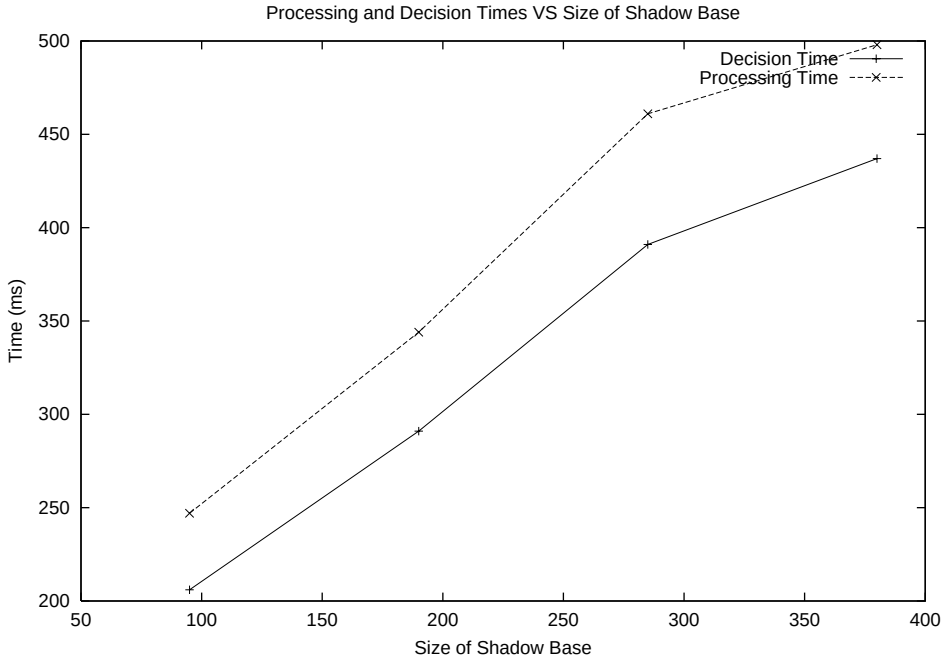


Figure 7-8: A graph showing the processing times and the decision times with respect to different sizes of the shadow base, for the case with facet pre-fetching.

Size	Without pre-fetching		With pre-fetching	
	% incr. (t_{proc})	% incr. (t_{dec})	% incr. (t_{proc})	% incr. (t_{dec})
95 $\rightarrow$ 190	55	50	41	39
190 $\rightarrow$ 285	28	26	34	34
285 $\rightarrow$ 380	21	19	12	8

Table 7.2: Percentage increase in the processing and decision times with and without facet pre-fetching.

and decision times increase with the size of the shadow base, the percentage increase becomes more gentle as the size of the shadow base becomes larger, as shown in Table 7.2. This suggests that when the size of the shadow base is large, a further increase in size does not cause the performance to degrade too much (i.e. it is quite scalable). This is a good sign for our proxy system that is designed to be used in pervasive computing environments.

Apart from scalability, the performance of functionality adaptation is also analyzed from the processing and decision times measured. From the results in the last two subsections, one observation is that the performance with facet pre-fetching is not improved as we expect. The reason is most probably due to the mis-prediction of the facet to be used. This mis-prediction is somehow inevitable because the run-time execution context cannot be correctly predicted. In order to have a better performance, better prediction needs to be explored. Despite that, the average performance is still acceptable. The average processing time for selecting a facet from 5-6 candidates is roughly 300ms (by taking the average of the processing times for a shadow base of size 190 in the two experiments); and the average decision time for making a decision among the same number of candidates is roughly 260ms (by similarly taking the average of the decision times in the two experiments). Usually, there are not too many candidate facets that can provide the functionality specified by the client, an average of 5-6 candidate facets can reflect the real situation for most of the cases. Although a processing time of 300ms cannot be said to be very efficient, it is already reasonable and good enough for the proxy system to be suitable for pervasive computing. Since performance optimization is not our aim in this prototype, it is left to be done later on.

7.4 Comparison with Random Selection

In order to have a better idea how well the functionality adaptation provided by our prototype performs, we further compare the adaptation with a random selection of facets for the clients. A request

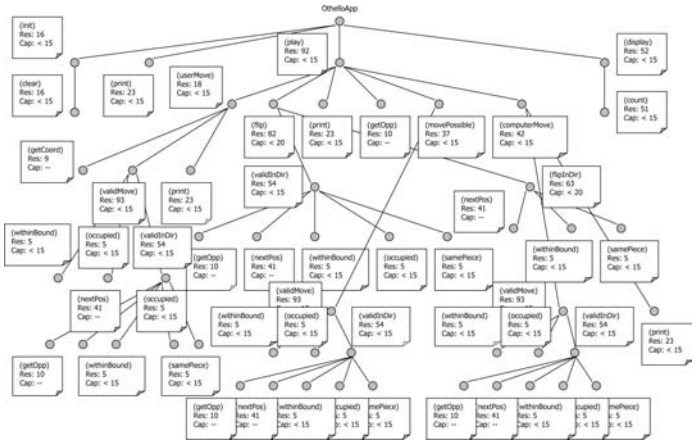


Figure 7-9: The facet execution tree using random facet selection.

is sent to the proxy server for matching facets that can provide the required functionality. It then randomly chooses from among the candidates a facet for returning to the client, without trying to make better decisions. The resulting facet execution tree for the requesting range $size < 10$ is shown in Figure 7-9.

The facet execution tree is quite different when compared with the one in Figure 7-4(a). The AQI calculated is 0.59, which is not as good as using functionality adaptation. With functionality adaptation, the quality is 1.2 times better. If user preferences can be better satisfied in functionality adaptation, the quality can be improved to 1.6 times better than the quality with random selection.

On the other hand, the performance using random selection is better because not much time is used in making the decision of a suitable facet to be returned. In the experiment, the average processing and decision times for selecting a facet from 5-6 candidates is about 198ms and 150ms respectively. which is a 50% speedup when compared to functionality adaptation.

In conclusion, although functionality adaptation can only achieve half the performance of random selection, the quality of adaptation is 1.6 times better. In a pervasive computing environment, execution contexts change quite frequently that makes context-awareness important so as to better satisfy the clients. In general, an average processing time of 300ms for functionality adaptation is not too bad, it is better to sacrifice a little bit of the performance in order to have a better facet to be adapted to the client.

7.5 Summary

In this chapter, we have evaluated the quality of functionality adaptation by giving the adaptation results of a few sample runs of an application. In order to visualize the quality of functionality adaptation, we have defined our own metric, called the Adaptation Quality Index (AQI), where a larger index value indicates a better adaptation of the service codes to the client. Besides evaluating the quality of functionality adaptation, we have also studied the proxy server's performance in functionality adaptation to analyze its suitability to be used in pervasive computing environments. In this aspect, the processing time and the decision time for returning a facet to the client are measured as a means of evaluating the performance.

From the results of these experiments, our prototype can be considered to behave well. The facets being selected by our proxy system, in general, have good qualities. If user preferences can be better satisfied, the adaptation quality can even achieve 86% from the optimal. For the performance, although the prototype is not optimized towards performance, a processing time of about 300ms can be achieved in most cases. This processing time is reasonably good enough for the proxy system to be used in pervasive computing environments. Further comparison with random facet selection indicates that functionality adaptation can only have half the performance but a quality of 1.6 times better. This sacrificing of the performance in order to have a better quality is acceptable in pervasive computing environments that requires services to be context-aware. With these experiments, our prototype is demonstrated to be suitable for functionality adaptation in pervasive computing environments.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The proliferation of mobile devices has changed people's way of computing. Computing is no longer performed in a fixed location, but can be done anywhere with the mobile devices. The concept of computing anytime, anywhere, and on any device has received much attention in the recent decade; and pervasive computing is now evolving to become the norm. The recent emergence of Web Services is a widely-accepted attempt to enable pervasive computing, where the heterogeneity of client devices is dealt with by relying on dedicated servers to provide services to clients. However, one computing model cannot suit all. There are cases where service codes should be able to be downloaded to the client devices for execution. In order to cater for the heterogeneity of the client devices used in pervasive computing, service codes should be adapted to the clients such that the desired functionality can be provided in the client devices. Functionality adaptation is, therefore, of great significance in pervasive computing environments.

In this thesis, we have presented a proxy-based approach for functionality adaptation in pervasive computing environments. We have also demonstrated the concept by implementing a simple prototype of a proxy system (or server) for functionality adaptation. A two-phase adaptation mechanism is used for the adaptation in the prototype. The first phase ensures that service codes can fulfill the requirements imposed by the clients, while the second phase allows a service code that best-suits the client in each request to be selected. The main difficulty lies in the first phase when service codes are checked to see whether they can provide the complete functionality in the client device. Since the dynamic resource usage of a piece of code varies from one execution to another, determining whether the code can be executed in the client device prior to the execution is a chal-

lenge. We have proposed a conservative prediction technique to estimate the total resource usage for executing a functionality. This conservative prediction makes use of the local maximum values of the dynamic resource usages of the codes that constitute a functionality, which are determined by the proxy server with information incorporated in each request.

We have tested our prototype for the quality and performance of functionality adaptation. Results have shown that the quality of functionality adaptation is greatly affected by the user preferences, which is usually difficult to be 100% satisfied due to the restriction placed by the functionality and resource requirements of the client. Despite that, with the quality of the adaptation being at around 65-70% from the optimal, the quality is still acceptable. If user preferences can be better satisfied, the quality can be improved to around 85%. In the other test, the performance for the proxy server in making a decision to return a suitable service code to the client is examined. The time taken to make such a decision is, on average, less than half a second. This shows that the decision making process is reasonably efficient. Results also show the scalability when the size of the shadow base increases. The reasonable efficiency and the ability to scale with the increase in the size of the shadow base show that the proxy server is suitable to be used for functionality adaptation in pervasive computing environments.

Proxy systems that are previously designed for adaptation to resource-constrained devices have only focused on adapting web contents. With functionality adaptation provided by our proxy system, clients are able to download service codes for execution, without worrying that the execution cannot be completed in the resource-constrained devices. This acts as a complement to the Web Services, and helps to enable truly pervasive computing by selecting service codes that are suitable for the clients to execute. Besides helping the normal device users, functionality adaptation can also help application/service programmers to simplify their efforts in building applications/services. Programmers can use service codes that are available for building their own applications/services. Instead of relying on the programmers to find suitable service codes for their use, they can simply put down some descriptions of the functionalities required, and allow the proxy servers to select suitable service codes to constitute to the application/service at run-time. With the use of capability matching, service codes of a greater capability can also be matched, and better service codes might be able to be selected to satisfy the clients.

Conservative prediction has contributed to the core of functionality adaptation so that service codes selected by the proxy servers can complete the functionality in the client device, provided that the service codes in the servers are not added or removed during the execution of a functionality and

the requests sent to the proxy server reflect the actual situations. However, conservative prediction also has its own limitation in that requesters of the functionalities are required to provide a brief idea about the range of the input size to be used at run-time. Without this information, conservative prediction cannot be done and the proxy server is unable to determine whether the service codes can provide the functionality in the client device. Nevertheless, with functionality adaptation provided by the proxy servers, clients should be able to enjoy computing anytime, anywhere, and on any device. We envision that functionality adaptation, together with the content adaptation techniques, can help to enable a truly pervasive computing for the near future.

8.2 Future Work

The proxy system that has been implemented is only a simple prototype to demonstrate functionality adaptation in pervasive computing environments. The prototype is only a minimal effort as a proof-of-concept and there are still room for improvements. Below is a list of future works that can be identified:

8.2.1 Best-effort Prediction

As we have mentioned, the main difficulty of functionality adaptation is in the adaptation of service codes to the resource requirement of the clients. In our conservative prediction approach, we have to rely on the requesters to provide an approximate information of the input size used at run-time. Although the requesters should be able to provide such an approximation (as they are the one to execute the service codes later on), it is more flexible if requesters are allowed to provide an infinite range when the size of the input data cannot be approximated. Despite that, the proxy server should still be able to predict the dynamic resource usage for executing a functionality and return service codes that can complete the functionalities in the resource-constrained devices.

By allowing infinite requesting ranges, we believe that proxy servers should not be able to guarantee the facets returned can complete the functionalities in the client devices. The best that the proxy server can do is to provide a “best-effort” prediction, such that each facet is predicted to use the maximum amount of resources available by the client. Since the resource usage of a facet is estimated by a resource formula that depends on the size of the input, such a prediction allows the maximum input size that can be used by the facet for its execution to be calculated. With this maximum size, the proxy server can have a fair comparison for the resource usages of different

facets. This is so-called a “best-effort” prediction because the proxy server tries its best to select a facet that can be executed with the client’s available resources. However, whether the functionality can be completed in the device cannot be guaranteed as it depends on the actual resource usage that has been used by the facet.

We have only provided a rough idea of the “best-effort” prediction at the moment. There are still some questions that need to be solved. For example, how this idea can be achieved in the real implementation? With a resource formula and the available resources, how can the proxy server determine the maximum size in real practice? Is there a better way for the “best-effort” prediction? All these problems need to be solved in order to allow infinite requesting ranges to be used.

8.2.2 Small Proxy Servers and Proxy Server Modules

At the current moment, the proxy server is assumed to run on resource-rich servers and have fixed connection to the Internet so that they can serve a larger number of clients at the same time. As peer-to-peer computing becomes more common, it seems to be a good idea if the proxy server can be made small enough to fit into resource-constrained devices. In that case, proxy servers can be run on small devices and be able to select suitable facets (although with some limitation) for nearby peers when the client is not connected to the Internet. This improvement allows pervasive computing to be achieved to an even greater extent: service codes can be downloaded from nearby peers in case of disconnection and to improve the access latency when compared to requesting the codes from the Internet.

On the other hand, in order for functionality adaptation to be easily incorporated into the existing Web model and act as a complement, it might be possible that functionality adaptation mechanisms are made into modules that can be incorporated into existing web proxies. In that case, web proxies can cache both the web contents as well as service codes, and be able to provide the required contents and codes to the clients. As a suggestion, Squid [44] seems to be a good platform for web caching, and incorporating the modules into Squid can be taken as the first step. To extend this further, the modules should also be able to be incorporated into transcoding proxies so that both content and functionality adaptations can be provided, which is good for enabling truly pervasive computing.

8.2.3 Improving the Facet Selection Scheme

In the current prototype, we have only used a simple scoring system for determining which facets to be selected. Basically, the scoring system in our prototype relies on artificial scores assigned to the facet if it has been cached/used before, as well as the scores indicated in the user profile that shows the importance of each preference to the user. The calculation of the scores is based on the assumption that user preferences are more important than proxy preferences, and a higher score is assigned to the facet where user preferences are satisfied instead of proxy preferences. It is possible to enhance the facet selection scheme for selecting better facets for the clients. For example, the proxy server may provide different facet selection schemes optimized for different situations, and it has the intelligence of selecting a suitable scheme suitable for the client according to their run-time execution contexts.

8.2.4 Enhancing the Prototype

The current prototype has not incorporated many supports (e.g. messaging to and from facet servers, security) that are needed for the proxy system to be used in pervasive computing environments. A complete prototype that incorporates all these supports are needed in the future. Besides implementing a complete prototype, optimization techniques are also needed in order to have a better performance in pervasive computing. Our current prototype uses a simple pre-fetching mechanism, where all the dependencies of the facet just returned and the first dependency of the next facet predicted to be used are pre-fetched. However, it seems that the performance in the experiment with the pre-fetching mechanism enabled is not too good. We believe that codes have a higher chance to be reused than data, and therefore should have a higher locality. This higher locality of code should provide a potential opportunity in improving the caching/pre-fetching performance. A better pre-fetching mechanism that is able to improve the performance should be explored.

8.2.5 Improving the Evaluation Metric

The evaluation of the quality of functionality adaptation is by the use of the Adaptation Quality Index (AQI) that we defined. This metric for evaluating the quality might be improved. For example, the time for returning a facet to the client might be taken into account by the metric as well. A smaller downloading time (which implies a smaller static size and hence, resource usage, of the codes) seems to favour. This downloading time has not been treated as a factor in the current

metric. However, the available bandwidth that affects the downloading time can be treated as the resource usage in the metric; and codes that consumes more total resources make better use of this resource and are treated as having better quality. Apart from downloading time, there might be other factors that can be added to the metric for enhancing the evaluation quality. In general, there are usually some trade-offs between different factors in a metric and should be carefully considered before adding them to the metric.

Bibliography

[1] Gregory D. Abowd, Anind K. Dey, Robert Orr, and Jason A. Brotherton. Context-Awareness in Wearable and Ubiquitous Computing. In *Proceedings of the 1st International Symposium on Wearable Computers*, 1997.

[2] Nalini M. Belaramani. A Component-based Software System With Functionality Adaptation for Mobile Computing. Master's thesis, The University of Hong Kong, 2002.

[3] Harini Bharadvaj, Anupam Joshi, and Sansanee Auephanwiriyakul. An Active Transcoding Proxy to Support Mobile Web Access. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[4] Timothy W. Bickmore and Bill N. Schilit. Digestor: Device-Independent Access to the World Wide Web. In *Proceedings for the Sixth International World Wide Web Conference*, 1997.

[5] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of Supercomputing Applications and High Performance Computing*, 14(4):317–329, 2000.

[6] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Technical report, Department of Computer Science, Dartmouth College, 2000.

[7] Yuk Chow. A Lightweight Mobile Code System for Pervasive Computing Environments. Master's thesis, The University of Hong Kong, 2002.

[8] Anind K. Dey and Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, 1999.

[9] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *IEEE Personal Communications*, 5(4):10–19, 1998.

[10] Fabiano Ghisla. The Jini Lookup Service: Services, Join, Lookup and Template Matching. A written report for the ..., December 2000.

[11] Teresa M. Griffie. The LAURA Experiment: Adapting Code to use the SSD on the Cray Y-MP. Technical Report RND-90-001, NASA Ames Research Center, 1990.

[12] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. A System Architecture for Pervasive Computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, 2000.

[13] Rob Horwitz, Michael Cherry, Paul DeGroot, Rob Helm, Peter Pawlak, Matt Rosoff, and Maurice Willey. Understanding .NET. Technical report, Directions on Microsoft Research, Inc., 2001.

[14] Timothy A. Howes and Mark C. Smith. A Scalable, Deployable, Directory Service Framework for the Internet. In *Proceedings of the International Networking Conference*, 1995.

[15] Anupam Joshi, C. Punyapu, and P. Karnam. Personalizaton & Asynchronicity to Support Mobile Web Access. In *Workshop on Web Information and Data Management*, 1998.

[16] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, 1998.

[17] Zhijun Lei and Nicolas D. Georganas. Context-based Media Adaptation in Pervasive Computing. In *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*, 2001.

[18] Chung-Sheng Li, Rakesh Mohan, and John R. Smith. Multimedia Content Description in the InfoPyramid. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1998.

[19] Naizhi Li. Supporting User Mobility with Web-based Mobile Computing. Master's thesis, The University of British Columbia, 2001.

[20] Steve Loughran. Towards an Adaptive and Context-Aware Laptop. Technical report, Hewlett Packard Laboratory, 2001.

[21] Vincent Wai-Yip Lum. Are Contextualization and Transcoding enough for Effective Content Adaptation Services for Mobile Computing? A survey paper.

[22] Rakesh Mohan, John R. Smith, and Chung-Sheng Li. Adapting Content to Client Resources in the Internet. In *IEEE International Conference on Multimedia Computing and Systems*, 1999.

[23] Rakesh Mohan, John R. Smith, and Chung-Sheng Li. Adapting Multimedia Internet Content for Universal Access. *IEEE Transactions on Multimedia*, 1(1):104–114, 1999.

[24] Herman Chung-Hwa Rao, Di-Fa Chang, Yih-Farn Chen, and Ming-Feng Chen. iMobile: A Proxy-based Platform for Mobile Services. In *Proceedings of the 1st Workshop on Wireless Mobile Internet*, 2001.

[25] James Rucker and Marcos J. Polanco. SiteSeer: Personalized Navigation for the Web. *Communications of the ACM*, 40(3):73–76, 1997.

[26] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[27] Narendra Shaha, Ashish Dessai, and Manish Parashar. Multimedia Content Adaptation for QoS Management over Heterogeneous Networks. In *Proceedings of the 2nd International Conference on Internet Computing*, 2001.

[28] João P. Sousa and David Garlan. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, 2002. To appear.

[29] Daby M. Sow, Guruduth Banavar, John S. Davis II, Jeremy Sussman, and Mugizi R. Rwebangira. Preparing the Edge of the Network for Pervasive Content Delivery. In *Proceedings of the Workshop on Middleware for Mobile Computing*, 2001.

[30] Ion Stoica, Robert Morris, David Karger, Francis M. Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM Special Interest Group on Data Communication Conference*, 2001.

[31] Son T. Vuong and Naizhi Li. WebMC: A Web-Based Middleware for Mobile Computing. In *Proceedings of the International Conference on Internet Computing*, 2000.

[32] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[33] Apache Content Negotiation. http://httpd.apache.org/docs/content-negotiation.html.

[34] Cybrant – Pervasive Computing Architecture. http://www.cybrant.com/tech/pervasive/.

[35] Ben Y. Zhao and Anthony Joseph. XSet: A Lightweight Database for Internet Applications. http://www.cs.berkeley.edu/~ravenben/publications/saint.pdf, 2000. Submitted for publication.

[36] How Domain Name Servers Work. http://www.scit.wlv.ac.uk/~jphb/comms/dns.html.

[37] Web Services Gotchas. http://www-106.ibm.com/developerworks/webservices/library/ws-gotcha/.

[38] IBM Think Research — Pervasive Computing. http://www.research.ibm.com/thinkresearch/pervasive.shtml.

[39] IBM Pervasive Computing. http://www-3.ibm.com/pvc/pervasive.shtml.

[40] IBM Transcoding In Depth. http://www.research.ibm.com/networked_data_systems/transcoding/In_Depth%/in_depth.html.

[41] IBM Web Sphere. http://www-3.ibm.com/software/info1/websphere/index.jsp?tab=highlights.

[42] DNS: The Domain Name System. http://www.rad.com/networks/1995/dns/dns.htm.

[43] Microsoft .NET Homepage. http://www.microsoft.com/net/default.asp.

[44] Squid Web Proxy Cache. http://www.squid-cache.org/.

[45] Sun Microsystems Open Net Environment (ONE) Homepage. http://wwws.sun.com/software/sunone/.

[46] External Annotation of Web Content for Transcoding. http://www.w3.org/TR/annot/.

[47] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP/.

[48] Web Services. http://www.w3.org/2002/ws/.

[49] one.world Homepage. http://one.cs.washington.edu.