October 2014

Proxy-based Mobile Computing Infrastructure

Azade Khalaj
The University of Western Ontario

Supervisor
Dr. Hanan Lutfiyya
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

PROXY-BASED MOBILE COMPUTING INFRASTRUCTURE

by

Azade Khalaj

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

In recent years, there has been a huge growth in mobile applications. More mobile users are able to access Internet services via their mobile devices e.g., smartphones and tablets. Some of these applications are highly interactive and resource intensive. Mobile applications, with limited storage capacity, slow processors and limited battery life, could be connected to the remote servers in clouds for leveraging resources. For example, weather applications use a remote service that collects weather data and make this data available through a well-defined API. This represents a static partitioning of functionality between mobile devices and a remote server that is determined at run-time.

Regardless of the network distance between the cloud infrastructure and the mobile device, the use of a remote service is well suited for mobile device applications with relatively little data to be transferred. However, long distances between a mobile device and remote services make this approach unsuitable for applications that require larger amounts of data to be transferred and/or have a high level of interactiveness with the user This includes mobile video communications (e.g., Skype, Face-Time, Google-Hangout), gaming applications that require sophisticated rendering and cloud media analysis that can be used to offer more personalized services. The latency incurred with this architecture makes it difficult to support real-time and interactive applications. A related problem is that the static partitioning strategy is not always suitable for all network conditions and inputs. For example, let us consider a speech recognition application. The performance depends on the size of the input and the type of connectivity to the backbone. Another challenge is that the communication medium between the mobile application and the remote service includes wireless links. Wireless links are more error prone and have less bandwidth than wired links. Often a mobile application may be disconnected.

One approach to addressing these challenges is the use of a proxy. A proxy is computing power that is located at the network edge. This allows it to address problems with latency. It is possible for a proxy to have services that allow for offloading tasks from either the cloud or the mobile device and to deal with communication challenges between the mobile application and the mobile device.

This work proposes a proxy-based system that acts as a middleware between the mobile application and the remote service. The proposed middleware consists of a set of proxies that provide services. The proposed middleware includes services for proxy discovery and selection, mechanisms for dealing with balancing loads on proxies and handoff. A prototype was developed to assess the effectiveness of the proposed proxy-based system.

Keywords: Mobile Computing, Mobile Cloud, Proxy, Cloudlet

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of mobile devices such as smart phones and tablets is ubiquitous. It is relatively easy to install and run applications on these devices. Some applications are well-suited for mobile devices e.g., Gmail, Google Maps, Yahoo stocks. Many of these applications communicate with a remote software service hosted on a server that processes data and/or collects data. Typically the data transferred between the mobile application and the remote service is relatively small.

Mobile devices have the potential to be suitable for many applications including e-health [24], e-learning [62], mobile TV [32], and augmented reality. These applications may be interactive or require large amounts of data transfer. However, the limited computing and storage resources on a mobile device limits the type of applications that can be executed on a mobile device. In this chapter we discuss limitations of current approaches to dealing with limited resources and provide a problem statement.

1.1 Offloading from Mobile Devices

One approach for enabling users of mobile devices to be able to use resource-intensive applications is to have part of the application run on remote servers, which may be part of a cloud infrastructure. The application component that executes on the remote server typically has high computational needs and/or requires access to data. These high computational needs are not easily available on the mobile device. The application component on the mobile device, which we will refer to as the *mobile application*, communicates with the application component on the remote server, which we will refer to as *remote service*. Typically the mobile application sends a request to the remote service and receives a result. The return of a result may be immediate e.g., a real-time translator. If the request is to receive a notification when a specific

event occurs then the result, i.e. a notification, is returned when the event occurs e.g., an email application that generates a notification when there is new email in the inbox.

Currently mobile applications represent a split of functionality that runs on the mobile device and on a remote server. The split is determined during application development. There are also methods that dynamically partition the applications. With dynamic partitioning, the two partitions, on the mobile device and on the server, dynamically change during program execution. The partitioning is done based on one or more factors. These factors include type of connectivity (e.g., WiFi or cellular), communication link load, or the load on the mobile device and the server. During the execution of the application some partitions might migrate from the mobile device to the server, or vice versa.

There are several examples of dynamic partitioning in the literature e.g., [23, 29, 55]. However, there is currently no example of employing dynamic partitioning for commercial applications. The reason for this is most likely due to the complicated methods used in the proposed dynamic partitioning approaches that add unacceptable time and computational overhead.

Communication between mobile applications and remote services takes place over wireless and wired links. Wireless links have lower bandwidth capacity than wired links. Furthermore, wireless links are more error prone. This results in data loss and disconnections. Although wired links are more reliable and have higher bandwidth capacity, the remote servers could be anywhere on the wired network and thus there could be high latency incurred for data transfers. The implication is that the model described earlier is effective when there is a small amount of data transferred between the mobile application and the remote service, and real-time interaction is not needed. Applications that are delay sensitive or involve a large amount of data to be transferred, such as applications involving video processing, are negatively impacted. Corradi et al. [22] argue that services offered by public cloud providers, such as Amazon EC2 [1], Windows Azure [13], Google Compute Engine [4], are not suitable for applications on mobile devices. The current model of partitioning an application between a mobile device and a remote server is not sufficient.

1.2 Using Proxies

To address some of the described challenges, the use of a proxy has been proposed [16, 31, 51, 60]. All communication between the mobile application and remote service passes through the proxy. In the literature, the proxy may range from a machine to a cluster. One proposed use of a proxy is to provide a diversity of services for mobile applications. These services may be used to ease the communication between the mobile application and the remote service. These services include disconnection handling, buffering and transcoding of multimedia data, etc.

Proxies are also proposed to reduce the communication delay between the mobile application and the remote service. Examples of systems employing proxies are presented in chapter 2.

The proxy concept is similar to other terms that have recently been introduced in the literature e.g., cloudlets [25, 60, 66], edge clouds [19, 39, 68].

1.3 Challenges in Using Proxies

Existing work shows that the use of proxies improves the quality of service provided for mobile applications. However, there are several challenges that need to be addressed to be able to make the use of proxies viable. In this section some of the challenges are presented and discussed.

1.3.1 Scalability of a Proxy-based System

In some of the work presented in the research literature, such as [21, 23, 44, 51, 60], there is an assumption that only one proxy is responsible for servicing all clients and that the access information of the proxy is known apriori by the mobile application. The use of a single proxy is not scalable as the number of mobile applications increases. In addition, the use of a single proxy introduces a single point of failure to the system. The presence of multiple proxies along with a load balancing mechanism addresses the limitations of using a single proxy.

1.3.2 Proxy Discovery

Factors that could affect the selection of a proxy for a mobile application, include the closeness of the proxy to the mobile device and to the remote service, the current proxy load, the security preferences of the mobile application and the mobile device e.g., if the provider of proxy is trusted or not. When multiple proxies are present, proxy selection will depend on the importance of specific factors. The importance of specific factors will depend on the users.

Some of the existing work on proxies discuss mechanisms for making available the access information of a proxy, or selecting a proxy from several available proxies. This includes the work that emphasizes on lowering the latency by employing a nearby proxy e.g., [21, 60].

There is a body of work (e.g., [25, 30, 58, 59]) that assumes that the mobile device should be connected directly to the local network the proxy machine is on. This means that if the mobile application wants to use the service provided by the proxy, it should stay connected to the same network until it no longer needs the proxy. To increase the flexibility with respect to associating mobile devices with applications, it is not desirable to assume that the proxies and mobile devices must be in the same network.

The *Cuckoo* system [40] assumes that the *Resource Manager* component of the mobile application has all the necessary information for all proxies. This does not scale if there is a large number of proxies since the information could consume a considerable amount of memory.

The literature shows that various factors are used in selecting a proxy. For example, available resources of proxies [34, 56, 57, 64], the cost of user/the revenue of service provider [48, 64], and the condition of communication links [54, 56, 65].

The distance between the mobile device and the proxy, and sometimes the distance between the proxy and the remote service, is an important factor when selecting a proxy for interactive mobile applications or mobile applications that involve the transfer of large amounts of data e.g. video to the proxy, or the remote service. Samimi et al. [58, 59] use the round trip time (RTT), as a measure of distance between the mobile device, the proxy and the remote service when selecting the proxy. The RTT is defined as "the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received" [9]. In Rajachandrasekar et al. [56] server nodes are chosen based on the RTT to the proxy.

Proxy discovery mechanisms proposed for mobile computing environment need to be scalable and should not limit the mobility of the mobile devices. It is also important to consider the condition of proxies e.g. current load on the proxy, the preferences of mobile application e.g. the time limits, and the condition of the wireless network, especially the distance between endpoints. None of the existing work considers a comprehensive view of proxy selection.

1.3.3 Static Association between Mobile Device and Proxy

A mobile device may move, the condition of wireless communications changes dynamically, or the proxy load may increase. A proxy that is initially suitable for the current environment and the requirements of the mobile application, might not be suitable after a period of time. As a result, even if the mechanism of discovering the most suitable proxy for the mobile application is defined in a system, it is crucial to monitor the conditions and change the associated proxy of a mobile client when needed. There is a need to dynamically associate mobile devices and proxies.

The mobility management approach presented by Bellavista et al. [17] defines the mobility as movement of the mobile device between domains. This approach does not take into consideration other factors such as changes in network or proxy conditions.

The mobility of the mobile device in Bellavista et al. [16] and Samimi et al. [59], which is defined as the change of IP address and the result of the mobile device leaving a network and joining another network, is supported by moving the proxy along with the mobile device to the new network. Deploying the proxy in the network that the mobile device is connected reduces

the communication latency and data traffic. Although, in general situations, the deployment of proxies in the local network of the mobile device might not be possible as the result of not having administrative access to the network that the mobile device is connected to.

The work proposed by Liang et al. [48] involves task migration between proxies. When migrating a task between hosts, there may be a need to transfer the state of the task as well. The tasks used in this work are stateless and there is no discussion on transmission of state.

The work described above provide some support for adapting to changes in the environment. However, support is limited. Methods are needed that do not add limitations, while providing adaptation mechanisms that respond to changes in the environment.

1.3.4 Specific-purpose Proxies

Most existing work assumes that the provider of the service is responsible for providing the proxy hardware and its services. Typically the services are installed on the proxy apriori. The implication is that the proxy is dedicated to a limited set of services that are pre-defined and makes the proxy a specific-purpose utility.

A more general approach is to decouple the responsibility of providing and maintaining the proxy system from the responsibility of providing services. Decoupling simplifies service provisioning for mobile applications by proxy providers and also improves the utilization of proxies, since a larger number of service providers will be able to exploit the resources available on a proxy.

1.3.5 Summary of Limitations

We address the following issues related to the use of proxies.

Proxy Scalability: In the majority of the research literature, there is an assumption that only one proxy is responsible for servicing all mobile applications and that the access information of the proxy is known apriori by the mobile application. The use of a single proxy is not scalable for a large number of mobile devices and the single proxy is a single point of failure. Multiple proxies and the use of load balancing approaches addresses the limitations of using a single proxy.

Proxy Discovery: The use of multiple proxies requires a mechanism for discovering and selecting proxies. Selection criteria include the closeness of the proxy to the mobile device and to the remote service, the current proxy load, security preferences of the mobile device e.g., if the provider of proxy is trusted for the mobile device or not, etc.

Dynamic Association between Mobile Device and Proxy: The association between the mobile device and a proxy should be dynamic. The mobile device may move, the state of the wireless communications medium may change or the proxy load changes. These changes could imply that another proxy should be used.

Static Proxy Services: In much of the related work, the proxy is designed and implemented to provide specific services. Typically a service is installed on the proxy apriori. The implication is that the proxy is dedicated to a limited set of services that are pre-defined.

Multipurpose Proxy: Most existing work assumes that the provider of the remote service used by a mobile application is responsible for providing the proxy hardware and software. This approach makes the proxy a specific purpose utility, and other services are not able to benefit from the resources made available by the proxy. A more general approach is to decouple the responsibility of providing and maintaining the proxy system from the responsibility of providing services. A proxy can provide generic services that are commonly needed by various remote services, such as disconnection handling or reducing latency. Subsequently, service providers can use services available on the proxies to service their mobile applications. The task of decoupling services from the proxy functionality, simplifies service provisioning for mobile applications by service providers and also improves the utilization of proxies since a larger number of mobile applications will be able to use the resources available on the proxy.

1.4 Contributions

This work addresses the limitations discussed earlier. We propose a proxy-based mobile computing infrastructure that tries to handle all of these requirements.

- The system includes multiple proxies, geographically distributed to enable the load distribution and also provide lower latency in the communication. The description of the proposed mobile computing system is presented in chapter 3.

- The use of multiple proxies requires a proxy discovery mechanism that finds the most appropriate proxy for a mobile application. The proxy discovery mechanism introduced for the proposed infrastructure is described is chapter 4.

- As described earlier there is a need to be able to change the proxy associated with a mobile application. This change requires a handoff mechanism that enables the change

of the associated proxy of a mobile application. The handoff mechanism designed for the system is introduced in chapter 5.

- The programming model introduced for the system simplifies the creation of mobile applications that use proxy services. The functionality of a proxy is modularized as independent services that can be installed/un-installed based on the requirements of the mobile application and the remote service. These independent services can be used by various services and mobile applications. The proposed programming model is presented in chapter 6.

- A prototype of the proposed infrastructure is implemented along with some example mobile applications to evaluate the effectiveness of the system.

Chapter 2

Literature Review

Several approaches and frameworks are proposed for mobile computing that employ a resourceful wired machine to empower mobile applications. In some application this machine is called the *proxy*, while in more recent work it is called the *cloudlet*. In fact both terms, i.e. *proxy* and *cloudlet*, in the mobile computing literature are used to name the same entity. In this work we use the *proxy* term, except for referring to, or describing the work that employed the *cloudlet* term. In this section we present some of more notable work in this area. The work is categorized into several categories based on the functionality of the proxy and services that a proxy provides in the proposed frameworks. We categorize the mobile computing frameworks into the following categories:

- Offloading computation from mobile device
  - Offload general computing tasks
  - Offload service specific task
- Reducing latency

This chapter presents representative work in each of the categories.

2.1 Offloading Computation

Executing compute-intensive tasks on behalf of mobile devices is the most common task of proxies in mobile computing frameworks. By offloading the compute-intensive tasks to a powerful proxy, the resources of the mobile device, including the battery power, memory and CPU cycles are saved, and also the capabilities of proxy is exploited by the mobile application. We divide the offloading proposals to two main categories. The first category includes the work

that proposes to offload the computation from mobile device, regardless of the functionality of the offloaded task. The work in the second category use proxies to provide a specific-purpose service for the mobile applications.

2.1.1 Offload General Computing Tasks

The work presented in this section usually involves code migration from the mobile device to the proxy.

2.1.1.1 MAUI

The MAUI system, presented by Cuervo et al. [23], involves a fine-grained offloading mechanism to the remote server. The system is designed with the goal of maximising the energy savings on the mobile device when running applications. It can be used for applications made to run on the Microsoft .NET Common Language Runtime.

The developer annotates the migratable methods. The MAUI solver component periodically is executed to find the best partitioning of the application between the mobile device and the remote server that minimizes the energy consumption on the mobile device. The characteristics of the device, methods of the application, and the network are measured by the MAUI profiler component. The input to the solver is provided by the profiler. The profiling of the device is done off-line, at the initialization phase and special hardware, not available on the mobile devices, is needed. The profiling of methods and the network is performed periodically during the program execution to find the up-to-date network shipping cost of methods to the remote server.

Two versions for migratable methods should be provided: one for local execution on the phone and the other for the remote execution. The latter version could be saved on a cloud storage and downloaded to the proxy whenever needed.

The MAUI framework provides a failure tolerance method. If a timeout occurs during the execution of a method on the remote server, the control returns to the mobile device. This could either invoke the method on the mobile device or find another MAUI server.

The methods can migrate from the device to the remote server during the runtime. As a result, it is necessary to find the state of the application and transfer the state between mobile device and remote server. A wrapper method is defined for each remote method that has one extra input parameter and one extra return value. The extra input parameter is used for transferring the application state to the proxy when offloading the method and transferring the new application state to the mobile device after finishing the method execution. In the MAUI system the application state is defined as the value of member variables of the current object,

including simple types and nested complex types. The state of all static classes of the application, along with the state of the public static member variables are included in the transferred state. The state is transferred using XML-based serialization.

The results of experiments that include the execution of four different applications shows that MAUI is able to save on energy consumption and improve the performance of applications by offloading partitions of application to the remote server.

Giurgiu et al. [29] proposed a system similar to MAUI for offloading partitions of applications from mobile devices to cloud resources. In their system the offloading unit is a module, unlike MAUI where the offloading is at the level of a method. In addition, in this work, the application initially is located only on the cloud and during the execution, some partitions of the application might be migrated to the mobile device to increase the application response time.

The Odessa framework [55] is another example of dynamic partial offloading from mobile device. Odessa combines partial offload with parallelism for *interactive perception* applications. Examples of *interactive perception* applications are face recognition, object and pose recognition and gesture recognition. The functionality of these application can be implemented as stages. The Odessa framework continuously decides which stages should be offloaded to the cloud to reduce the response time and increase the accuracy of application.

2.1.1.2 SuMMIT

The SuMMIT (Submission, Monitoring and Management of Interaction of Tasks), presented by Rosseto et al. [57] is a framework, which controls the execution of applications that are composed of multiple tasks and submitted from mobile devices to the available servers. A GUI component is installed on the mobile device that allows the user to submit requests to the framework and check the status of execution of tasks included in the submitted application. The other parts of the framework reside on a machine on the wired network which plays the role of a proxy.

A component named the *Resource Selector* uses an ontology-based approach for describing resources, policies, and requests to perform matchmaking for received requests. The authors claimed that their approach is flexible for adding more characteristics for resources, however in the provided use case, the characteristics used to describe the resources are available memory, processor capacity, installed software, and available databases. The request coming from mobile application should specify the number of servers that are needed, along with the required values for the above characteristics.

The proxy is responsible for matching available resources of servers with the requirements

of tasks submitted by the mobile device, and dispatching tasks to servers. The proxy is also responsible for collecting results from the servers, which execute dispatched tasks, and send the results to the mobile device. The *Agent* component on the proxy is responsible for monitoring the connection to the mobile device. If the *Agent* finds that the connection to a mobile device is broken, it decides whether the submitted tasks, belonging to the disconnected mobile device, should be continued, stopped, or aborted, based on the options set by the user of the mobile device. Options are declared in a file submitted to the framework on the proxy, as part of the user's request.

If the user's option is to continue the execution of the application in the case of a disconnection, the results obtained after the completion of the application will be saved on the proxy to be delivered to the mobile device after the reconnection. If the user's option is to stop the execution of the application after discovering a disconnection, the application execution will be stopped and the state of the application will be saved in a checkpoint file on the proxy. When the mobile device connects again to the proxy, the proxy will notify the mobile device about an uncompleted application. If the user of the mobile device decides to continue the execution of the application in case of the disconnection, the state of application is loaded from the checkpoint file and the execution of the application will be continued.

2.1.1.3 Economic Mobile Cloud Computing

An economic mobile cloud computing model is presented by Liang et al. [48]. The system is composed of multiple clouds with different providers. The goal of a cloud service provider is to maximize its resource utilization and also its revenue. The model takes into consideration the cloud computing capacity, the overall cloud system gain, and expenses of mobile users using cloud services. The cost of a mobile device is defined in terms of battery consumption and the cost of using cloud services. It is assumed that several cloud providers exist in the form of different cloud service provider domains.

Mobile applications are implemented as *weblets*. *Weblets* can be executed on the mobile device or can be migrated to the cloud and executed on the cloud. On a cloud, the service nodes (SNs) are responsible for managing the loading or unloading of *weblets*. Each SN handles only one *weblet*. A mobile device can migrate its *weblets* to a cloud by contacting an SN on the cloud. The SN decides to accept the migration request, or transfer it to another domain. On the cloud, *weblets* can be executed on one or more virtual nodes in the domain of a cloud provider.

An SN accepts three types of requests: i) new - a new *weblet* migration request from a mobile device, or transferred from another cloud provider domain, ii) intra-domain - an existing *weblet* transferred from another SN in the same domain, iii) inter-domain - an existing *weblet*

transferred from an SN on another domain.

The transfer of *weblets* to another domain means a loss of revenue for a domain. As a result, a domain tries to maximize its resource utilization and accept more migration requests from mobile devices. The authors presented a Semi-Markov Decision Process (SMDP) economic model to find the optimal resource allocation, and maximize the gain of cloud providers and reduce the expense of mobile user. The most revenue is obtained from the intra-domain transfer of *weblets*, rather then from accepting new *weblets*. Inter-domain transfer means revenue loss and a cloud provider tries to avoid it.

2.1.1.4 Cuckoo

The Cuckoo framework, introduced by Kemp et al. [40], is targeted at the Android platform. When developing Android applications, it is possible to separate the UI activities from computational activities. The Cuckoo framework provides a programming model that allows the computational part of an Android application be offloaded to a remote resource. The remote resource that runs the offloaded task can be on a commercial cloud or a private computer. The remote resource should have the JVM available, as well as a Java application called the *server* which is used to run tasks offloaded from the phone.

The Cuckoo framework is integrated with Eclipse and parts of the application development are automated. The Cuckoo framework helps the developer to develop the local and remote implementation of the application and bundles them in one package. Eclipse builders provided by the Cuckoo framework are used for this purpose. The local implementation of an application should be created by the developer. The Cuckoo framework uses the local implementation to create the template for the remote implementation. It is the responsibility of the developer to implement the methods of the remote implementation. The local implementation can be different from the remote one to allow the application to use resources available at the resource-rich cloud, such as multiprocessor capability.

A component, named the *Resource Manager*, on the mobile device, maintains the information about registered remote resources. Before running the compute-intensive activity, the Cuckoo framework decides whether to offload the services or not. The Cuckoo framework contacts the *Resource Manager* to see if any registered remote resource is accessible. If there is a remote resource available, the application is offloaded to it. The jar file of the application is sent to the remote resource and the remote resource installs the application. After the installation, the client on the phone can call offloaded methods of the application. The methods that interact with sensors of the phone, such as camera, should be marked by the developer to never be offloaded.

The Cuckoo framework only supports the offloading of stateless tasks and no state transfer is done. If after offloading a task to the remote resource a timeout occurs and no response is received from the remote server, the Cuckoo framework assumes that the remote server is not accessible anymore and tries to find another remote resource and offload the task to it. If there is no available remote resource, the service is executed locally on the phone.

An object recognition application and a face detection application are used to evaluate the performance of the Cuckoo framework.

2.1.1.5 Interdroid

Interdroid, a framework for integrating mobile application with the cloud, introduced by Kemp et al. [41], is created based on the Cuckoo framework (section 2.1.1.4). Authors have shown the challenges in mobile-cloud integration using a scenario and have realized several integration problems.

Employing the proposed approaches for offloading the computation from mobile device to the cloud, results in the situation that the mobile device users are dependent on the cloud providers. As a result, when there is no access to the part of the application running on the cloud, the application turns out to be unusable.

The first problem is that applications of mobile-cloud environments are designed to have some components running on the mobile device and some components on the cloud. The communication between components on the mobile device and cloud has cost and sometimes there is no connectivity available. The authors believe that all components should be available as well on the mobile device, and the application should be configurable so that the user has the ability to decide which components run on the mobile device or on the cloud.

The second problem is that if the application is divided into two parts, i.e. the client and the service, while the client runs on the mobile device and the service runs on the cloud, the mobile device user has no control of the service component. If the cloud owner turns out to be untrustworthy e.g., degrades the service quality or increase the price, the mobile device user can not do anything. The authors propose to have both client and service components of the application bundled on the mobile device. Thus the user always has the ability to install the service part on another cloud and use it.

The third problem is related to accessing data. The authors believe that the user should be able to get data on and off the cloud. This means that the user will be able to move to another cloud without being locked into a contract with a cloud. In addition, the local replication of data should be available whenever no connection is available to the cloud.

To address the first and second problem the Cuckoo framework is employed. The work in

Cuckoo is about bundling the client and service in one package that the user owns. In addition, the decision is made at runtime to run the service part locally, or on the cloud.

There is a class of mobile applications that need to periodically poll software services for getting up-to-date information e.g., weather, traffic, stock market information. This polling consumes large amounts of battery power on the mobile device. The authors proposed to provide cloud services that can be used for polling the required services and if there is a change in the information, the service pushes a notification to the mobile application on the mobile device. These services are also bundled with the client part of the application and the user has the ability to install the service on any cloud.

The authors introduced a project named Raven that is a framework for synchronizing and replicating data between mobile device and the cloud. This framework is designed to give the control over data to the user of the mobile device. While the data is kept on the cloud, the mobile device can replicate part of data locally when is online and keep data to use even when the devices goes offline.

2.1.1.6 Offload to a Cluster

The work described by Rajachandrasekar et al. [56] allows a smartphone to ask for the execution of a compute-extensive applications on remote computing resources. The remote computing resource is a cluster of machines based on the Alchemi framework, which is a .NET Grid computing framework. The application used to evaluate the proposed system calculates the first n digits of the number Pi.

A PC is used as a proxy. The PC listens for requests coming from the mobile device containing the parameters needed for running the task. Based on the input parameters, the proxy divides the task into threads. The PC proxy then finds the most suitable of available resources on the cluster and distributes threads among them. The proxy waits for the completion of all threads, collects and combines the results returned from the cluster nodes and sends the final results to the mobile application.

It is assumed that the task is located on the proxy and the mobile application on the mobile device is only responsible for providing the input parameters. The functionality of dividing the task into threads, finding the best resources and submitting the threads to the cluster node should be available on the proxy.

2.1.1.7 Energy Saving: Local Processing vs. Offloading

The effect of offloading to the cloud from the mobile device based on the energy consumption is also explored by Kumar et al. [47]. The paper presents an analysis to show how the wireless

bandwidth, the amount of transferred data, and the processing power of the remote server impacts the energy consumption when the computation is offloaded from the mobile device to the cloud.

The paper shows that if the amount of transferred data is small and the wireless bandwidth is large enough, it is beneficial to offload computation from the mobile device. The result is based on the assumption that there is always a high processing capacity available on the cloud. The cloud can also be used as a storage to save the user data and, as a result, the amount of data transferred between the mobile device and the cloud can be reduced.

For applications that need to have access to real-time data, such as image retrieval based on the newly captured images, the paper proposes pre-processing on the real-time data on the mobile device. The results of pre-processing are then transferred to the cloud to reduce the size of transferred data.

2.1.1.8 A Proxy on the Local Network

The work proposed by Guan et al. [30, 31] uses a proxy to offload compute-intensive tasks from resource constrained mobile devices to more powerful resources. A module on the mobile device, named the *Device Proxy Module*, which works based on the Universal Plug and Play (UPnP) protocol, is responsible to discover nearby proxies upon entering a local network.

After finding a proxy, the mobile device can send a request to run an application to the proxy. If the proxy has enough resources, it downloads application code from an online repository, installs and runs it. Otherwise, the proxy invokes the related service on the remote server hosting the requested service. After completion of the application, the proxy sends the result to the mobile device.

To keep requests of different mobile devices independent from each other and also to make the management of requests easier, the virtual machine technology is used on proxies.

Guan et al. [31] argued that context-awareness is an important factor in pervasive applications which is one of the applications that needs the infrastructure of the mobile computing. An ontology-based model is proposed in Guan et al. [31] to describe the context.

2.1.1.9 Chroma

The Chroma system proposed by Balan et al. [15] splits applications between the mobile phone and a remote server. The system is designed to enable the developer to modify existing applications for being partitioned. The Chroma takes into consideration the preference of the user, in terms of the latency and the accuracy, when splitting the application.

For each application several tactics are recognized. Each tactic shows a way of splitting the application between the mobile phone and the remote server. Tactics are added manually to the application by the developer. At runtime, Chroma chooses the best tactic based on the availability of resources and the resource usage. There is a need to find the resource usage of each tactic to choose the best tactic. The resource usage patterns of tactics are calculated by recording the resource usage of each tactic during the execution of the application.

The resource availability on the mobile phone is measured by the Chroma system. The resources measured include memory and CPU usage, available bandwidth, network latency and battery power level. The system uses a utility function to choose the best tactic. The utility function is defined as the ratio of accuracy/latency and the goal is to maximize the utility function.

The authors proposed to use over-provisioning to improve the performance of system. The over-provisioning is provided by replicating the same task on several remote servers and using the result from the fastest remote server. The other way of over-provisioning is to partition the input data between multiple remote server and integrating the results received from all servers.

The example applications used for evaluating the Chroma system are a translator, a speech to text and a face recognition applications. The results show that the Chroma sometimes has not chosen the best tactic because of the inaccurate initial values for the the resource usage of tactics. In addition, the time needed for choosing an appropriate tactic increases drastically when the number of tactics increases.

2.1.1.10 Improving the Reliability Using Replication

To provide higher reliability in mobile computing Trung et al. [64] proposed to offload a task to multiple proxies. Delegating a job from a wireless device to a proxy could fail for several reasons, such as overloading of the proxy, a failure in the proxy hardware/software or disconnections in the wireless link.

To increase the reliability, this work proposes to replicate a submitted task and send it to several proxies which are called Replication Gateways (RGs). A primary RG is responsible for accepting requests. This responsibility includes collecting information about the requested service from the mobile device, choosing RGs, and submitting tasks to chosen RGs. The RGs try to discover the requested service and send the request along with the input data, received from the mobile device, to the service. Upon completion of the request and having the result, RGs send the result back to the mobile device. If one of the gateways fails, the mobile device can still receive the result from other gateways.

A higher number of RGs means higher reliability. However, it comes at a higher cost. The

authors provided an algorithm which is used to calculate the number of RGs based on the user's budget and the cost. The cost is calculated based on the cost of creating a new replica.

2.1.1.11 Energy-aware Broker

An energy-aware scheduler for the mobile computing is proposed in Park et al. [54]. A centralized scheduler, residing on a machine called the *Broker*, decides whether a mobile device has enough battery power to execute a task or not. If not, the scheduler decides to which server node the task can be migrated and executed on. The scheduler is aware of the cost of the execution of a task on the mobile device and on all server nodes, in addition to the migration cost of the task from mobile device to the server nodes. The information about the the hardware characteristics of the mobile device and the residual energy at the mobile device should be known by the scheduler.

2.1.2 Offload Specific Services

The work presented in this section introduces the use of proxies that provide services for specific tasks, such as multimedia streaming. In this work, it is commonly assumed that services are installed beforehand on the proxies and mobile applications on the mobile devices use those services for accessing to specific remote resources.

2.1.2.1 Elastic HTML5

The work presented in Zhang et al. [67] proposes to distribute the execution of web browsers, using HTML5 API, between the mobile device and cloud. The work uses the *web worker* component introduced in HTML5. A *web worker* is a script that runs in the background within the browser. It can communicate with the HTML page. Every *web worker* has its own database to save the web page related information. In Elastic HTML5 *web workers* are enabled to be executed on the mobile device or on the cloud. The database of a *web worker* can reside on the mobile device or on the cloud as well. *Web workers* should be able to migrate transparently between mobile device and the cloud.

A pair of specifically designed *web workers*, as proxies, are used for the communication between components of web application on the mobile device and the cloud. The proxy *web worker* on the mobile device is called the Elastic Web Worker Manager (EWWM). The proxy *web worker* on the cloud is called the Cloud Web Worker Service (CWWS). Whenever an HTML5 application invokes a *web worker*, the EWWM decides on whether to launch it on the mobile device or on the cloud. If the EWWM decides to execute it on the cloud, it requests the

CWWS by transferring the code of the *web worker* and the parameters. The CWWS launches the transferred *web worker* and returns the end-point address of the *web worker* in the form of a URL to the EWWM. When the *web worker* wants to create its database for the first time, the database is created locally, regarding the location of the *web worker*.

To enable the operation of the web application during offline periods, a device-side database that is kept synchronized with the cloud-side database can be used.

In this work it is assumed that the cloud can be either a home/enterprise cloud or a public cloud.

2.1.2.2 Self-Adaptive Data Link Layer Handoff Manager

The work described by Bellavista et al. [16] introduces an application layer infrastructure for self-adaptive handoff management in the data link layer of the OSI model [8]. The authors proposed a proxy service for mobile devices that is specifically made to manage the data link layer handoff at the application layer. The data link layer handoff happens between access points of the mobile device to the wired network. The proposed infrastructure can be used for horizontal handoff, when two access points work based on the same radio technology, as well as the vertical handoff, when handoff happens between two different radio technologies e.g., WiFi and Bluetooth.

The interruption as the result of the handoff has a negative impact on the quality of the multimedia content streamed from a server to the mobile device. One way to eliminate the delay added by the handoff is by having a larger buffer at the mobile device. However, the mobile devices often do not have enough memory for a large buffer. Also, this method does not work for live multimedia streams. To solve this problem, using an additional buffering proxy between the streaming server and the mobile device is proposed. Large buffers can be kept at this proxy. The proxy is usually installed close to the mobile device.

Another way to mitigate the negative impact of the handoff, is to employ soft handoff. During a soft handoff, the mobile device continues to be connected to the original access point until the connection to the target access point is ready. At that time, the mobile device starts to use the connection to the target access point and releases the connection to the original access point. It is unlike the hard handoff that as soon as the handoff starts, the connection to the original access point is released and the mobile device waits for the completion of the connection to the target access point. The soft handoff is faster than the hard handoff, but it consumes more battery power.

The proposed infrastructure includes a proxy server. The proxy server decides if the handoff should be a soft or a hard handoff. The proxy server, based on the condition and the specifi-

cation of the multimedia service, decides if a hard handoff is sufficient or a soft handoff is needed. The paper allows multimedia service providers to specify the requirements of their service. This service specification is used during the handoff management process. The specification of multimedia service is expressed as the expected quality, using the Video Quality Metric (VQM) standard and the expected delay. The value of metrics along with values taken from the mobile device about the condition of network connectivity are used to calculate the size of buffer at the proxy. If the server proxy decides to do a hard handoff, the system decides on the size of the buffer at the proxy server. By increasing the size of buffer, the system tries to mitigate the impact of the delay because of the handoff.

2.1.2.3 i5 Multimedia Cloud Architecture

Kovachev et al. [44] argue that mobile devices generate lots of multimedia content. Mobile applications for processing multimedia content are highly specialized and expensive. It is not easy for amateur users to use those applications. On the other hand, cloud computing technology focuses on scalability of businesses and not on users. As a result, the facilities provided in the cloud computing are not suited for requirements of users of mobile devices.

The paper provides a framework for developing multimedia applications for mobile devices combined with cloud computing. The authors proposed three perspectives that a mobile multimedia platform should consider: technology, mobile multimedia, and user and community. Based on the requirements that arise from these perspectives an architecture is designed for mobile multimedia.

The platform, named i5 Multimedia Cloud Architecture, consists of four layers: infrastructure layer, platform layer, multimedia services layer, and application layer. The infrastructure layer is provided by a private cloud. The virtual machine technology is used in this layer to separate the hardware from software. The platform layer consists of engines that are used by the services in the upper layer, multimedia services layer, as well as Deltacloud API [2]. The Datacloud enables the system to provide access to public cloud facilities along with the access to the private cloud.

The paper used the cloudlet concept, introduced by Satyanarayanan et al. [60], to decrease the latency by hosting some services on a nearby private cloud. The multimedia services layer includes services that are commonly used by various multimedia applications, such as Semantic Metadata services, Collaborative Multimedia services, Transcoding, Bitrate Adjustment, etc. Finally the last layer, Application layer consists of applications that are made by users, exploiting the services in the lower layer.

Three applications are implemented to evaluate the performance of the proposed platform.

Applications are used for sharing multimedia content and metadata. A user can record video, annotate it and send it to services provided by the platform. The video is transcoded and other video processing, such as object detection, is done by multimedia services available on the platform. The video can then be streamed to other users that can modify or add new annotation to the video online.

2.1.2.4 DYNAMO

Mohapatra et al. [51] propose a proxy-based real-time transcoding of multimedia data that is streamed from a multimedia server to mobile devices. The system tries to find a trade-off between power consumption and the quality of the video played on the mobile device. The proxy is located close to the mobile device and is responsible for buffering and transcoding. All communication between mobile device and media server passes through the proxy server.

The system is made of a middleware layer, distributed on the mobile device and the proxy server. The middleware part on the proxy uses end-to-end system state e.g., network noise, mobility, and proxy server load, along with the feedback received from the mobile device, to decide about video transcoding and the network traffic shaping. Admission control is done on the proxy. Based on the feedback from the mobile device, the proxy decides if a request for a multimedia streaming service can be scheduled or not. The proxy performs the admission control based on the request of the mobile device, including the minimum video quality accepted by the user, the residual battery power, and the length of the video. The proxy is also able to buffer data and regulate the data transmission to the mobile device with the purpose of reducing power consumption on the mobile device. When the proxy adapts the video transcoding, the middleware on the mobile device might adapt some settings on the mobile device, such as the backlight, the CPU and network adapter sleep time.

The system has the “local state awareness” as well as the “global state awareness”. The “local state” is the condition on the mobile device, i.e. residual power, backlight setting, CPU and memory information. The “global state” is the information not available in the mobile device, i.e. bandwidth availability, network condition and the mobility information. The mobile device is responsible for monitoring the “local state” and reporting it to the proxy. The proxy is responsible for determining the “global state”.

A system utility function is defined based on the energy consumption on the mobile device and the quality of the video played on the mobile device. The goal is to maximize the quality of video by choosing the appropriate transcoding while taking into consideration the residual battery power of the mobile device.

2.1.2.5 MAPGrid

Huang et al. [34, 35] propose to use the idle resources available in the system, to stream multimedia content from repositories to mobile devices. The introduced middleware has three components: the mobile device, the *Broker*, and the *Volunteer Server (VS)*. *VSs* volunteer to dedicate their idle resources, i.e. storage and communication resources, to this multimedia streaming system. *VSs* download segments of the requested multimedia file from repositories and after transcoding the multimedia content to suit the capabilities of the requesting mobile device, transmit it to the requesting mobile device. The set of *VSs* to fulfill the request of a mobile device is chosen by the *Broker* which maintains information about available *VSs*. Upon receiving a request from a mobile device, the *Broker* tries to create a schedule to stream the requested multimedia content to the requesting mobile device, based on the availability of *VSs*. If the *Broker* cannot create a schedule, it rejects the request. Otherwise, the *Broker* informs *VSs* about the schedule and the segments that the *VSs* are responsible for downloading and transcoding.

This middleware is power-aware and takes the device specification into consideration. The mobile device should announce the level of its residual battery power, along with its request to the *Broker*. The *Broker* uses this information to decide if the mobile device is able to receive the requested service or not. If yes, the *Broker* decides the level of the quality of the transmitted multimedia based on the residual battery power and capabilities of the mobile device. This means that if the mobile device has more capable equipment or has a higher level of energy available, it will receive a better quality content. The *Broker* and *VSs* monitor the conditions and dynamically change the schedule. If a *VS* notices that the load of tasks on the host machine has increased and the host machine is not able to serve some of the requests, it can notify the *Broker* and request the migration of some tasks to another *VS*. The *Broker* also monitors the activities of *VSs* and in the case of a failure in a *VS*, it arranges a new schedule to complete the processing of the request of mobile device. If the *Broker* cannot find a new schedule, rejects the request.

The MAPGrid introduced by Huang et al.[36, 38] is an architecture based on [34] and [35]. It uses techniques that take into consideration the intermittent availability of resources at *VSs* that are called proxies. A two-tier design is proposed. One tier is responsible for determining the availability and capacity of proxies, and the other tier is responsible for determining the pattern of data requests sent by mobile clients. By having this information, an optimal mapping between proxies and segments of the requested multimedia file can be calculated. This mapping changes dynamically with changes in the availability of proxies. To evaluate the performance of this system three metrics are used. The first metric is the average distance of the mobile

client to proxies in terms of network hops during a service period. The second metric is the data replication cost which is done to download segments of multimedia file from a repository server to the proxies selected by the the *Broker* to accomplish a service. The last metric is the percentage of load saved on repository server in comparison to the situation where there is no *Broker* and no proxies.

2.2 Reducing Latency

Access to services provided by the proxy adds network latency to the interaction between the mobile application and the service which is not acceptable especially for delay sensitive applications. This section describes some of the work that focuses on mitigating the negative impact of network latency.

2.2.1 CloneCloud

The overall description of the CloneCloud framework is presented by Chun et al. [20]. The CloneCloud is designed for the mobile phone applications that runs on a VM, such as Android applications that run on the Dalvik VM. In this system the entire smartphone image is cloned on a proxy that can be a nearby desktop or laptop computer. The application is automatically partitioned and the computational intensive part of application is offloaded to the proxy, along with its state. Upon the completion of the offloaded task, the state of the offloaded task is sent to the primary application on the phone and is merged with the state of the primary application.

The detailed description of the system, including finding the migration points in the application, building the cost model, collecting the state, and performing the migration is presented by Chun et al. [21]. The granularity of migration in the CloneCloud is at the thread level. A computationally heavy thread can be offloaded to the clone on the cloud, while other threads of an application continue to run concurrently on the phone. If a thread running on the phone needs to access the state of an offloaded thread, the thread on the phone will be blocked until the offloaded thread finishes and returns the new state to the phone.

Native system operation can be used both on the phone and on the clone running on the cloud, provided that the underlying OS on the cloud permits. This characteristic of CloneCloud allows the phone application to exploit the capabilities of the cloud resources that do not exist on the phone, such as high-bandwidth network and powerful API's available on the cloud.

The task migration in the CloneCloud is done using application level VM migration. The migration of a thread in the application level includes transferring the instruction set of the thread in the form of byte codes along with the state of the thread. The state of a thread

includes virtual stack, virtual registers, and the native stack of the thread. To find the possible migration points, an offline partitioning process is done. In the offline phase, the *static analyzer* finds the legal migration points. The *dynamic profiler* then profiles the application to find cost models for both running the application on the phone and on the cloud. The profiler is executed several times for different input values and different conditions to generate a database of various partitionings and their cost models for different conditions. At runtime, the condition of the system, including the availability of cloud resources and the network bandwidth, is looked up in the database of partitionings and an appropriate partitioning is chosen.

To evaluate the CloneCloud, authors modified the Dalvik VM to enable it for partitioning and migration of Android applications. Three applications are used for the evaluation: a virus scanner, an image search, and a privacy-preserving targeted advertising. The *static analyzer* is run on a desktop computer and takes on average 20 seconds which is fairly long. However, this static analysis is done once for each application. The experiments are designed for running applications on the phone and on the cloud. To offload to the cloud, both WiFi and 3G network connectivity are used. Three different sizes of input data are tested for all applications. Experimental results shows that the partitioner decided to offload the computational heavy thread of all applications when the size of input is larger, but it decides to run the entire application locally on the phone for small input sizes. The speedup and energy consumption for all offloading cases is improved over local execution except for one case that offloading happens over the 3G network. The results achieved from runs on the WiFi network surpass all results taken on the 3G network.

2.2.2 Transient Customization of Cloudlets

The use of cloudlet is proposed by Satyanarayanan et al. [60]. The proposed approach employs local clouds, named cloudlets, to decrease the communication latency when processing is offloaded from mobile device to the cloud. The cloudlet is hosted on the LAN that the mobile device is connected to. The VM technology is used to offload computation to the cloudlet.

The authors claim that mobile devices will remain resource poor since the producers typically work on improving the size, weight and battery power. On the other hand, the obstacle for using cloud in mobile computing is latency. The delay in communication over WANs hurts the usability of applications. Interactive applications are suffering even with moderate RTT of 100 ms. The authors claim that the latency of WANs will not improve in future since network providers focus on other targets, i.e. security, bandwidth, energy consumption, etc., while achieving improvements for these targets will increase the latency. The bandwidth is also limiting for applications that need to transfer bulky data, such as image/video processing

applications.

The framework introduced in the paper includes cloudlets that are decentralized and dispersed. Cloudlets can be installed in coffee shops or doctors' offices and it is possible to integrate them with the WiFi access points. To prevent code or data loss, cloudlets use cached copies of code and data that are saved somewhere else. The proposed infrastructure is named the"*transient customization of cloudlets*". *Transient* means that a pre-use customization, along with a post-use cleanup is done for each mobile application. Employing the VM technology helps to encapsulate each mobile application on the cloudlet. In addition, it does not restrict applications to be implemented in a specific language.

The method for VM state delivery is the dynamic VM synthesis. A VM overlay is created by the mobile application and is transferred to the cloudlet. The base VM, that the overlay is derived from, already exists on the cloudlet. The cloudlet applies the overlay to the base VM and launches the VM. Afterwards, the client application on the mobile device starts to interact with the launched VM. When the mobile application is done, the cloudlet creates the VM residue and sends the VM residue to the mobile application and discards the VM on the cloudlet.

The authors state three advantages of their system. The proposed system i) is suitable for interactive applications because of the closeness of cloudlets; ii) reduces the bandwidth demand, specially for multimedia applications; and iii) is adaptable to the requirements of various applications, because of using VM technology.

The time between transferring the VM until resuming the VM on the cloudlet is high, i.e. between 60 to 90 seconds. Authors have mentioned some methods for improvement. Authors have addressed several open issues: i) who provides the cloudlets, business owners or service providers, ii) how security and privacy are preserved, and iii) how much resources should be provided by cloudlets.

2.2.3 Interactive Applications and Cloudlets

A system of multiple cloudlets for the interactive mobile cloud applications (IMCA) is proposed by Fesehaye et al. [25]. The authors assume that future applications will be interactive in nature. As a result it is more efficient to install services on the nearby cloudlet, rather than accessing services on the public clouds.

The work in the paper involves a network of cloudlets that cover a specific area, called the *proximal community network*, and service interactive mobile applications on mobile devices in that area. Each cloudlet can support multiple mobile devices. Cloudlets are equipped with wireless access point hardware and mobile devices in their vicinity can connect to them in one

hop. A routing protocol among cloudlets is proposed and cloudlets can support peer-to-peer communication between mobile devices. To use the cloudlet-based system, a mobile device should find a nearby cloudlet. The mobile device decides to use a cloudlet or contacts the cloud directly. To make the decision, the delay to the cloud and the cloudlet, or the bandwidth of the path to cloudlet and to the cloud are measured and compared. The mobile device chooses the one with smaller delay, or higher bandwidth. It is assumed that if the mobile device decides to use its closest cloudlet, an instance of the needed service, is already available in the cloudlet, and the needed data, which is kept on a cloud, can be downloaded to the cloudlet.

While using a cloudlet, if the mobile device moves and enters the area covered by another cloudlet, the mobile device connects to the nearby cloudlet. The newly connected cloudlet relays messages between the mobile device and the servicing cloudlet.

The authors compared the performance of the proposed cloudlet-based system with direct communication between mobile device and the cloud. The scenarios are simulated using NS2. Three applications are examined: file editing, video streaming, and chat. For the file editing application, the file to be edited is downloaded from the cloud to the cloudlet. After finishing the edit, the file is updated on the cloud by the cloudlet. For the video streaming application, the video is downloaded from the cloud to the cloudlet. The streaming of video from cloudlet to the mobile device can start before the completion of download from cloud. The chatting service exists on the cloudlets and the proposed routing protocol lets the mobile devices to find their intended chat target, i.e. another mobile device, in the area covered by the cloudlets.

The movement of mobile devices leads to getting connected to other cloudlets and it increases the hop number between mobile device and the servicing cloudlet. Results of simulations show that if the mobile device is steady, or if the mobile device moves, but the hop count of cloudlets between the mobile device and the servicing cloudlet is equal or less than two, using cloudlets outperforms using services on the cloud directly. Although, if because of the movement of mobile device, the hop count increases to more than two, it is better to use the service on the cloud.

2.2.4 Partial Offload to Cloudlet

The work proposed by Verbelen et al. [66] is based on the Satyanarayanan's cloudlet infrastructure [60], introduced in section 2.2.2. Authors proposed that instead of migrating the entire VM from the mobile device to the cloudlet, only some partitions of the application be migrated.

In the proposed system, the cloudlet is created dynamically using devices available in a LAN. Instead of transferring the whole VM from the mobile device to the cloudlet, a static partitioning of application is done and the CPU intensive partitions are offloaded to the cloudlet.

To be able to use the resources of cloudlets, mobile applications should be created in the form of partitions, and offloadable partitions should be marked by the application developer. The resource allocation on the cloudlet is dynamic. If the cloudlet runs out of resources, the cloudlet can offload less delay-sensitive partitions of application to a public cloud.

The authors proposed an architecture for their *cloudlet* system and implemented it. The architecture includes three entities: *component*, *node* and the *cloudlet*. The *component* is the unit of job that can be offloaded from mobile devices. The *node* is the hardware and its installed OS. Multiple *nodes* can join to form a *cloudlet*. Each *node* is managed by a *Node Agent* and each *cloudlet* is managed by a *Cloudlet Agent (CA)*. The CA is responsible to decide about the installment of *components* on the *nodes* of the *cloudlet*. Some experiments are done for an AR application.

The authors proposed a decision algorithm in Verbelen et al. [65] for the distribution of *components* of application among members of the *cloudlet*. The parameters taken into account for the decision making are the network condition, i.e. bandwidth and delay, available resources on *cloudlets* and constraints of *components* that are defined by the application developer.

2.2.5 Integration of Public Clouds with Cloudlets

The authors in Bahl et al. [14] note that today's public clouds are not suitable for mobile applications and tried to answer how public clouds should change to be suitable for mobile applications. Currently, many mobile applications use various services on the public clouds. The problem is that there is duplication in the functionality of these cloud services, for example location awareness, mobility management, and application partitioning between mobile device and the cloud. The authors propose to modify cloud computing so that those common tasks be provided as part of the cloud.

Additionally, authors believe that offloading to the closest data center does not provide low latency, while offloading over a LAN will be much better. They propose the installation of cloudlets on the "wireless access network", WiFi hotspots, and peer mobile devices. It is also proposed to seamlessly integrate cloudlets with the closest data center of a public cloud, and at the same time specialize the infrastructure of cloud for mobile computing. The requirements for this proposal are i) the presence of high speed VPN between the datacenter and the cloudlet, ii) support of high performance VM migration from cloudlet to the public cloud when cloudlet is resource-less, iii) the availability of the public cloud to augment the resource level of cloudlet, and iv) storing a copy of persistent data on the public cloud.

The authors propose that cloud providers should provide three types of services for mobile applications: i) platform services, such as databases, storage and compute nodes, ii) application

services, such as speech/image recognition, video streaming and push notification, iii) Context-rich services, such as context extraction, recommendation service and group privacy service.

2.2.6 Pocket Cloudlet

In Koukoumidis et al. [45] the claim is that even if the bandwidth of the cellular network increases, latency will remain a bottleneck. As a result, transferring data between mobile devices and the cloud will remain a challenge. It is proposed, since the memory size is increasing on mobile devices, to save the data needed for mobile applications on the mobile device. The advantage of this approach is the reduction of latency and battery power consumption, the personalization of services, and the preservation of privacy.

The requirements for having such a facility on the mobile device are the following: i) being able to determine which part of data should be saved on the mobile device since the mobile device is not able to keep all data available on the cloud, ii) keep the data on the mobile device updated, iii) Having a storage architecture on the mobile device for efficiently storing and accessing the data.

For the data selection, authors proposed to choose data based on a personal and community model. For keeping the data updated on the mobile device, it is proposed to update static data when the mobile device is charging, weekly or monthly. Although for dynamic data, data should be updated real-time, over radio links. For fast access to data on the mobile device a memory architecture is proposed. A search application is used for evaluating the proposed system. To apply the proposed architecture for an application it is needed to analyze the data required by the application to find if data is cachable or not. Authors argued that data for search application is cachable. Results of experiments for the search application shows speedup when using WiFi and 3G and also reduction in energy consumption.

2.2.7 Proximal Workspace

Taylor et al. [63] proposed a development platform for the mobile devices that can be used for AR applications. The system is made of three types of components: the *terminal*, the *proximal workspace*, and the *world*. *Terminals* are lightweight mobile devices that are used for input/output purposes. The *world* is the set of AR services that are accessible through the Internet. The *proximal workspace* is responsible for executing computation/data intensive tasks on behalf of the *terminals* and calling the services in the *world*. The *proximal workspace* is located in proximity of the *terminal* to decrease the communication latency.

A sample application, used to evaluate the effectiveness of the system, is introduced in the paper. The application is the *Google Earth Ancient Rome 3D*. The service is installed on

a remote server, while the client of this service is installed on the *proximal workspace*. The client application takes input data from the *terminal*, then forwards it to the Google service. The video frames sent by the Google service to the mobile application on the *proximal workspace*, are then forwarded to the *terminal*. The experiments show that the lag between frames is eliminated when the Google client app is executed on the *proximal workspace*, comparing to the scenario that the mobile application is running on the *terminal*.

The authors specified that the *proximal workspace* in not expected to be able to run arbitrary code, but will offer a set of gadgets that have been developed to handle common tasks.

With the approach presented in this work, the client of a cloud service can be installed on the *proximal workspace*, while a new application should be implemented for the *terminal*. The client application on the *proximal workspace* should be altered to take the input data from the application on the *terminal*, although, the remote service stays intact.

2.2.8 Data Placement on Nearby proxies

Huang et al. [37] proposed to use proxies for localized data placement for mobile applications. The advantages of the localized data placement are the following: i) reducing the load on the server, ii) reducing the latency when the mobile application accesses data on the proxy, since the proxy is closer to the mobile device, and iii) reducing the overall load on the network.

The proposed data placement procedure, Reconfigurable-Mobile Data Overlay (Rec-MDO), uses the aggregated information of all users' data accesses in the past, to provide a long-term data placement for the future requests. The authors argue that since the mobility of the mobile devices are co-related with location and time, the aggregated information about past usage can be used as an indicator for future data accesses by users.

It is assumed that the data accessed by mobile devices are video content. Data segmentation techniques are used. This means that the video is divided into multiple objects, and each object can be replicated on different proxies.

The goal of the Rec-MDO procedure is to achieve a balance between the access cost and replication cost, and also to minimize both costs. Every T units of time the placement procedure is executed and decides about replicating data objects on proxies. When assigning data objects on a proxy, the available resources of the proxy, i.e. CPU, storage, and network bandwidth, are taken into consideration.

A Binary Search Tree data structure is used to save the data access information of users. The nodes of the tree saves the information for time intervals. Each node saves the number of accesses to each data object, on each proxy, during the corresponding interval. At the end of each period, the data in nodes of the tree are updated, and the placement procedure is executed.

The procedure, based on the similarity of the access to contiguous objects, might decide to merge contiguous objects and replicate them on a single proxy. In other situations, the procedure might try to find another proxy for replicating a data object. This is done to reduce the access cost to the data object by taking into consideration the cost of replicating the data object to the proxy.

The proxy chosen for replicating the data object should have enough resources. A proxy can be chosen in a greedy manner, i.e. choosing a proxy with enough resources and minimum replication and access cost. Alternatively, load distribution among proxies can be preserved when choosing a proxy.

The system is simulated and several experiments are done that shows the Rec-MDO procedure is successful in reducing the replication and access costs.

Chapter 3

Proxy-based System

Proxies in the mobile computing solutions are widely used in literature. Usually, compute-intensive tasks are offloaded to the proxies. Proxies have employed for other responsibilities, such as mobility handling, providing secure communication, reducing latency, etc.

This chapter includes the description of the architecture of our proposed proxy-based system and a discussion of the role of the architecture in applications.

3.1 Proposed System

In this work a proxy-based infrastructure for mobile applications that includes multiple proxies, that are geographically distributed, is introduced. The use of multiple proxies allows for a distribution of workload. To reduce the communication latency, the distance between the proxy and the remote service or between the proxy and the mobile device can be taken into consideration, when assigning a proxy to a mobile device. The system is also able to react to changes in the network environment.

As presented in chapter 2, uses of proxies include offloading computation from mobile devices, facilitating the communication between remote resources and mobile applications, and also reducing the communication latency. Proxies provide a set of services that we refer to as *proxy services*. Proxy services are independent of the functionality of remote services and mobile applications. This allows proxy services to be used by different remote services and mobile applications. Each proxy can install/un-install *proxy services* based on the needs of mobile applications, and its available resources. As a result, the set of proxy services that a proxy provides can dynamically change over time.

Since multiple proxies are available in the system there is need for a mechanism that allows a mobile application to discover and select a proxy. This mechanism must be relatively light

weight since mobile devices do not always have an abundance of computing resources. The *Proxy Finder Server* (*PFS*) component, introduced in section 3.2.3, is responsible for activities related to discovering proxies. This requires that proxies communicate with the *PFS*.

The use of *PFS* means that a mobile application does not need to hard code access information of the proxy. The proxy discovery mechanism finds the most suitable proxy for a mobile application in response to a request from the mobile application. The request includes information related to the requirements that the mobile application expects the proxy to satisfy. This includes network latency, trust information and response time. In addition to the requirements, the state of the proxies are taken into consideration when selecting a proxy. The details of the proxy discovery mechanism is presented in chapter 4.

The proxy associated with the mobile device may change in response to changes in proxy load, available bandwidth on communication links and changes in the location of a mobile device. The proposed system is able to automatically adapt to changes, whenever needed, to keep the service quality at a satisfactory level. This is done by changing the associated proxy of a mobile application. This process is called the *handoff process*. The details of the handoff process and the conditions that initiate the handoff process are presented in chapter 5.

The description of the components of the proxy architecture and their roles are presented in section 3.2.

3.2 Components

This section describes the primary components of the proxy-based architecture.

3.2.1 Proxy

In this work proxies provide a set proxy services. Proxy services can be used to offload a computationally heavy task from mobile devices, download an interactive service from a cloud to a proxy in order to reduce communication latency, improve the reliability of the communication between mobile applications and the remote service by providing support for handling disconnections or transcoding, etc. Each proxy can dynamically install/un-install proxy services when needed.

There are two models for the use of proxies. In the first model, the proxy is a gateway between the mobile application and the remote service. Communication between the mobile application and the remote service is through the proxy. Figure 3.1 depicts a use of this model. In figure 3.1 the proxy has two proxy services: PS_1 and PS_2. These services communicate with a remote service. Assume that proxy service PS_1 provides failure handling. The use of a proxy

service that handles failures is illustrated with the following scenario. The connection between a mobile application and a remote service may terminate prematurely due to a software or hardware failure. The request from the mobile application to the remote service may be lost or the result from the remote service to the mobile application could be lost. When the connection is re-established the request has to be re-sent. A proxy service can be used to relay the request and save the result returned from a remote service. When the mobile application re-establishes the connection to the proxy it can retrieve the result without having to send another request to the remote service.

The second proxy service, PS_2, could be used for downscaling of multimedia data. This proxy service requests a video from the remote service on behalf of the mobile application and transcodes the video to an encoding more appropriate for the capabilities of the mobile device.

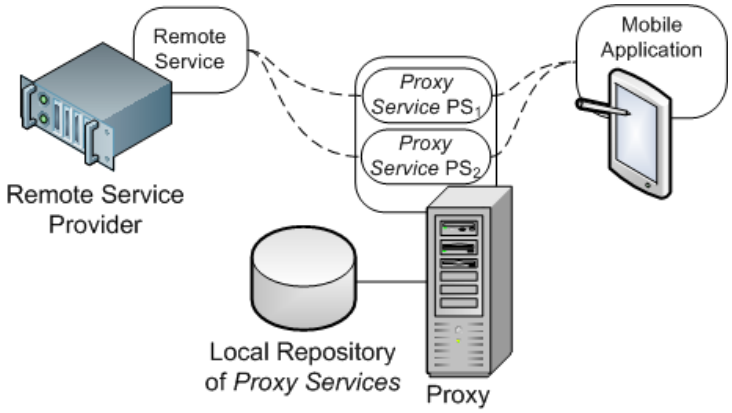


Figure 3.1: Proxy Use - Model 1

In the second model, which is illustrated in figure 3.2, the communication is only between the mobile application and the proxy. No call to a remote service, hosted on another server, is made by the proxy. In figure 3.2, the mobile application uses one proxy service, PS_3. This proxy service could be used to offload computation from the mobile application to the proxy for execution on the proxy.

Mobile applications typically assume communication with a remote service that provides a well-defined API that can be used by the mobile application. This represents a static partitioning of an application where part of the application runs on the mobile device and another part runs on the server. However, the best application partition may depend on the network conditions. For example, let us consider an object tracking application. The performance depends on the size of the input and the type of connectivity to the backbone. Since WiFi is generally faster than cellular, if the connectivity to the backbone is through WiFi, more data can be transferred than if the connectivity is through cellular. This suggests a need to be able to partition an application at runtime and then offload the selected partition to the proxy. This proxy service

can be used to execute the application component placed on the proxy and provide the mobile application with the result of the execution.

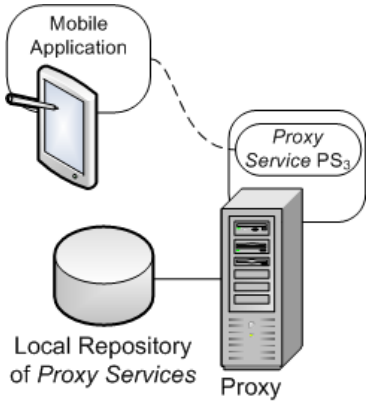


Figure 3.2: Proxy Use - Model 2

3.2.2 Proxy Services

As seen in chapter 2, proxies can be used to provide a set of services. A proxy service can be independent of remote services and mobile applications. This allows for the reusability of proxy services.

Each proxy service provides an API that allows mobile application developers to develop applications that make use of the *proxy services* to access remote services or resources available on a proxy.

When a proxy service is used to access a remote service, the proxy service receives the requests of mobile applications and input data, which are relayed to the remote service. The remote service processes the request and sends the result back to the proxy service. Subsequently, the proxy service relays the result to the mobile application. Essentially, the proxy service plays the role of the client for the remote service.

Wireless communication links have low bandwidth and frequent disconnections in accessing remote services. Remote services that require a high level of interaction or require the transmission of large amounts of data is not always viable for use by mobile applications. Proxy services can be used to provide specific support for mobile devices when calling those remote services. Since the remote service views the proxy service as the client, the remote service can be left non-altered. This advantage enables the remote service providers to implement customized proxy services that can be specifically used by the mobile applications to access their remote services without making any change to the remote services.

3.2.2.1 Example Proxy Services

This section provides several examples of proxy services.

Handling Failures. A failure may occur during a call to a remote service. This failure may occur before the mobile application receives any results. Causes of a failure include a disconnection between the mobile device and the access point to the wired network, or the failure of the mobile device due to low battery power. A proxy service can be designed to buffer and relay data between the mobile application and the remote service to mitigate the effect of failures. The mobile application sends the request and input data to the proxy service and the proxy service calls the remote service on behalf of the mobile application. After receiving the result from the remote service, the proxy service relays the results from the remote service to the mobile application. In the case of a connection failure between a mobile application and the remote service, the proxy can buffer the result sent from the remote service in response to a request from the mobile application. After recovering from the failure and establishing a new connection, the proxy service can send the buffered result to the mobile application. In addition, the proxy can send keep alive messages to the remote service, during the period that the mobile application is disconnected, to keep the connection up, until the mobile application is done with the remote service. This service provided by the proxy is highly valuable for disconnection-prone wireless environment and for mobile devices with limited battery storage.

Reducing Latency. It is important to have low communication latency when executing interactive or real-time applications. Example applications include gaming, multimedia, and augmented reality (AR) applications. A proxy service can be provided for the purpose of reducing the communication latency. This proxy service, installed on a proxy in the vicinity of the mobile device, can download the requested remote service to the proxy, install and launch it. The vicinity can be defined as the hop number between the mobile device and proxy, or the RTT between the mobile device and proxy. After installation of the remote service on the proxy, the remote service can respond to requests sent from the mobile application. This decreases the communication latency between the mobile application and the service, installed on the nearby proxy. Bahl et al. [14] is an example of work that uses proxies to reduce the communication latency.

Downscale Multimedia Data. Some services involve video streaming or image transmission. Mobile devices with relatively little computing resources may not be able to handle high bit rate video/image encodings. Low bandwidth wireless links also slow down the transmission

of high bit rate encodings. The proxy can provide a proxy service to downscale the multimedia data, provided by the remote service, then stream or transfer it to the mobile application. This proxy service can transcode the multimedia data to a suitable transmission bit rate and format, based on the physical specification of mobile device, and the characteristics and the transient condition of the path to the mobile device. Examples of work that use proxies to provide multimedia services for mobile applications are found in [34, 44, 51].

Offload Computation. As a result of limited resources available on a typical mobile device it may be feasible to offload compute-intensive tasks to the proxy from an application on the mobile device. A proxy service on the proxy can be provided to process offloaded tasks locally, or it may distribute tasks to another server with available resources. In the literature, related to the mobile computing, a good deal of work is about offloading the whole or a part of computational extensive tasks to the proxy. The work proposed in [23, 40, 57] are a few example of work using proxies to offload computation from mobile devices.

Adapt Services for Mobile Computing Environment. Some remote services are not aware of the limitations of client applications that are hosted on mobile devices. These remote services are usually designed with the assumption that clients are running on powerful desktop machines and are connected to high-bandwidth wired links. The remote services that need low communication latency e.g., applications that require a high volume of data transmission or are highly interactive (e.g,. AR applications), often cannot work with acceptable quality for clients on mobile devices. One solution could be to redesign and re-implement these remote services to be able to take into consideration the capabilities of their clients and the communication environment, and adapt their services appropriately. Instead, a proxy could be employed to avoid the expenses imposed by the alteration of the remote service. Proxy services can be designed that adapt the service provided by the remote service to the characteristics of the mobile computing environment. Example proxy services include transcoding, buffering, or pre-fetching the data. These proxy services are mostly specific-purpose and should be provided by the remote service provider that aims to provide its mobile clients with higher quality. Taylor et al. [63] is an example of work that provides support for mobile applications to use an existing remote service with the help of proxies.

3.2.2.2 Provider of Proxy Services

A proxy service can be generic, i.e. it is designed to answer a common requirement of mobile applications, and can be used by various mobile application to access different remote services.

The proxy services that handle disconnections or are used for computation offloading are examples of generic proxy services. Generic proxy services can be provided by developers and uploaded to online repositories for access by mobile application developers.

Some proxy services have a specific purpose, i.e. they are created to enable mobile applications to communicate with specific remote services. The proxy services designed to adapt available remote services to the mobile computing environment, are examples of specific-purpose proxy services (e.g. Taylor et al. [63]). Providers of commercial remote services might be providers of these specific-purpose proxy services.

A remote service provider may choose to have its own specific proxy services because because of security and privacy concerns. A remote service provider may do not want to give access permission to its services from generic proxy services.

3.2.2.3 Finding Proxy Services

Proxy services can be kept in online repositories that are accessible by the proxies. The address of an online repository should be provided to proxies beforehand. As a result, when a proxy is used by a mobile application that requires a specific set of proxy services, the proxy can download the missing requested proxy services from the online repositories. The proxy can keep the most wanted or newly used proxy services in a local repository to eliminate the download time for future requests for those proxy services. The service providers that have their own specific-purpose proxy services might have their own repositories.

3.2.2.4 Deployment of Proxy Services

Each proxy can host multiple proxy services. A mobile application that is designed to take advantage of a set of proxy services, needs to find a proxy that provides all of the required proxy services. Alternatively, a proxy that does not have all the required proxy services, but is able to download and install the missing proxy services, can be used by the mobile application. The details of the process of finding an appropriate proxy is presented in chapter 4.

To use a proxy, a mobile application sends a request to the proxy. The mobile application requires the contact information of an appropriate proxy, e.g. IP address and port, apriori. The proxy discovery mechanism is used to find a proxy and returns an IP address to the mobile application. The request, sent to the proxy, includes, among other data, the list of *proxy services* needed by the mobile application. Afterwards, the proxy downloads the required proxy services that are not in its local repository, and then installs all the required proxy services, which are not installed yet. After installing proxy services, the mobile application can start communication with the proxy services.

When the mobile application requests a proxy service, an instance of the requested proxy service is created for the mobile application. In this way more that one mobile application can access an installed proxy service on a proxy.

When no mobile application is using any instance of a proxy service, the proxy can un-install that proxy service to free resources. Un-installing of a proxy service includes un-installing it and removing data sources used by the proxy service, such as databases, directories, etc. In fact proxies behave like a dynamic library of proxy services that change based on the demand of mobile applications.

3.2.3 *Proxy Finder Server (PFS)*

A *PFS* is responsible for finding a proxy for a mobile application upon request. Criteria for proxy selection includes the proximity of the proxy to the mobile device, proximity of the remote service to the proxy and proxy load. The weight to be given to a criterion is based on preferences of the mobile application.

The address and other information about proxies, such as available proxy services of proxies, and the information regarding the trustworthiness of the proxies, that are needed during proxy discovery phase, are kept by the *PFSs* in a repository. The mobile application that wants to use the proxy-based system, should have the address of at least one *PFS*. The details of proxy discovery process are presented in chapter 4.

PFSs and proxies cooperate to handle issues related to the mobility of mobile devices, changes in proxy load, and network condition such as communication latency. A proxy monitors the conditions e.g., the communication delay or the CPU load on the proxy. If the monitored data does not satisfy certain conditions, the proxy and *PFS* carry out the handoff operation that includes finding a more appropriate proxy for the mobile application and handoff the mobile application to the newly found proxy with less interruption in the activity of the mobile application. The details of the handoff operation are described in chapter 5.

For the purposes of scalability, elimination of a single point of failure, and load balancing there could be multiple *PFSs* in the system. Proxies can register at multiple *PFSs*.

Since *PFSs* may have the information of different sets of proxies a mobile application can query more that one *PFSs* for proxy discovery. It will speed up the proxy discovery process. It is also possible that the mobile application chooses the most appropriate proxy from the proxies found by various *PFSs*.

The network of proxies and *PFSs* can be provided and maintained by a third party. Possible third parties include an ISP, telecom company, or a company that is a provider of the proxy-based infrastructure. The service providers can have their private network of proxies and *PFSs*

and provide their custom made proxy services for their mobile clients.

The address of *PFSs* should be provided for proxies and mobile applications (the same way that the address of Domain Name System (DNS) servers are provided for network devices). There are multiple mechanisms that can be used to provide mobile applications and proxies with addresses of *PFSs*. For example, *PFSs* addresses could come from an ISP provider or a telecom company.

3.2.4 Client Side Component

Mobile applications need to be extended with a component to use the proxy-based infrastructure and access to proxy services. The component is a library that is designed to enable the mobile application to find a proxy, by sending a request to the *PFS*, and calls the *proxy services* instantiated specifically for this client on the proxy. The library uses the API of the *PFS*, proxy, and proxy services.

3.2.5 Remote Services

Proxy services are used to support mobile applications to facilitate communication with the remote service. These proxy services provide functionality for the mobile application to call a remote service. For example, the *Downscale Multimedia Data* proxy service, introduced in section 3.2.2, transcodes the multimedia data, streamed from a remote multimedia server, to an encoding suitable for the mobile device specification and the network condition e.g., cellular vs. WiFi connectivity or traffic load.

The remote services are not part of the proxy-based infrastructure and can be any kind of services, such as services made available on a public cloud, or services that are available on the private cloud of a service provider. As mentioned in the section 3.2.1, not all proxy services are used for accessing remote services.

3.3 Using the Proxy-based System

This section illustrates through examples the relationship between proxy services, mobile applications and remote services. Section 3.3.1 introduces a remote printing application, and section 3.3.2 describes a tourist assistance application that uses augmented reality technologies. The proxy services used for the example applications are introduced in each of the above mentioned sections. Finally, for each application we show how the mobile application can use the proposed proxy services to access remote services.

3.3.1 Remote Printing in a Campus

The remote print service enables mobile users to remotely submit their print jobs to the closest available printer in a university campus. The remote print service consists of two services: the *Printer Finder* and the *Printing Service*. The *Printer Finder* selects the best printer based on user preferences e.g., location, and printer specifications. This service can be made available on a public cloud or on the private cloud of the university. This service needs to have access to the database containing the information of the printers and users. The information about users might include the role (student, professor, etc.), or affiliated department. The accessibility rules of printers are also saved in the database. Only a subset of printers may be accessible by users having specific roles, or the printers belonging to a department may be accessible only by the students, professors, and staffs of the department.

One of the factors in finding a printer is the closeness of the printer to the location of the user. This metric is defined by the user's preference. It can be based on the geographical distance or the networking delay between the mobile device and the printer. If the user prefers a printer within the closest geographical distance, the service finds a printer in the physical vicinity of the current location of the user, using GPS coordinates. If the networking delay between the printers and mobile device has more importance for the user, the *Printing Service* associated with the printer is able to use the *ping* tool to find the RTT between the mobile device and the printer. In some cases the documents to be printed is large, and the communication latency has a noticeable impact on the transfer time of the document. In these cases, the user might prefer a printer that has the smallest communication delay with the mobile device.

The *Printing Service* is used for printing a document on a remote printer. There is an instance of the *Printing Service* for each printer. The reason for having two separate services is that the user may have already selected a printer and thus has no need for the *Printer Finder*. Both services provide an API for use by mobile applications.

To use the remote printing service, the user launches the related mobile application on the mobile device. If the user uses the *Printer Finder* service, then the user should specify his/her preferences and other required information. The user's preferences and proxy information is used to find the the best printer for the user. The contact information of the *Printing Service* associated with the best printer is returned to the mobile application.

If the user wants to use a specific printer, there is no need to call the *Printer Finder*. The user can provide the address of the target printer e.g., by choosing the printer from a list of printers.

In both cases, the user should specify the location of the document, to be printed, on the mobile devices. Afterwards, the mobile application contacts the *Printing Service* to accomplish

the printing task.

3.3.1.1 How the Proxy-based System Helps This Scenario

The time between the mobile application invoking a call to the *Printer Finder* service and receiving the result could be several seconds. The reason is that if the user's preference of the distance measure is the RTT, the *Printer Finder* service waits for each *Printing Service* to determine its RTT to the mobile device. To speed up this operation, the *Printer Finder* can send requests to all *Printing Services* simultaneously. If there is a large number of *Printing Services* to contact, it might not be feasible to contact all of them simultaneously because of the limitations in the available bandwidth or the limitation in the number of concurrent running threads. In that case, the *Printer Finder* would divide *Printing Services* into groups and contact *Printing Services* in each group at a time. As a result, a larger number of *Printing Services* might lead to a longer response time for the *Printer Finder* service. Even if the *Printer Finder* service is able to call all *Printing Services* at the same time, receiving responses from *Printing Services*, might takes several seconds because of the way that the *Printer Finder* works. The *Printer Finder* may use the ping tool by sending multiple probes, in intervals equal or larger than one second, to have a better estimation of the current RTT of the route. In this way finding the RTT to a *Printing Service* may take multiple seconds.

The other possible factor for measuring the distance between the mobile device and the *Printing Services* is the hop count. A greater hop count between the two components means longer communication latency since there are additional routers and each router does some sort of processing. Using this factor, the closest *Printing Service* to the mobile device is the *Printing Service* that has the smallest hop count to the mobile device. The disadvantage of using the hop count, as opposed to using the RTT for calculating the distance, is that the hop count is fixed and does not show the changing condition of the path between two components. The condition of the path is affected by the changing traffic distribution on the path. In wireless links, the condition is also affected by the interferences. Unlikely, the RTT factor shows a snapshot of the condition of the path, since the traffic load or the interferences in the wireless links affect the RTT.

While the mobile application is waiting for the result from the *Printer Finder*, it is possible that the connection of the client application on the mobile device to the *Printer Finder* service is terminated as the result of mobile device movement or some other communication interruption. The mobile application should call again the *Printer Finder* service and wait for another several seconds.

Another difficulty occurs when the mobile application tries to transfer a large document to

a remote *Printing Services* for printing. If no additional support is provided for uploading large files by the *Printing Service*, this file transfer may be interrupted several times and the mobile application may have to retransmit the file after each interruption.

3.3.1.2 Proxy Services for Remote Printing

Three proxy services are needed to support the functionality of the remote printing applications. The proxy services are described as follows.

***Relay* Proxy Service** A remote service may need to carry out a time-consuming action. This includes processing large amounts of data, querying a large database or a call to other remote services. When a mobile application on a mobile device calls a remote service that executes time-consuming operations, it is possible that a disconnection occurs between the mobile application and the remote service before that mobile application receives the result of the service call. In this case, if there is no mechanism to support the mobile application, the mobile application needs to wait for a re-connection to the remote service. The *Relay* proxy service can be placed on a proxy, and all the messages between the mobile application and the remote service, including the request and the result, are passed through the *Relay* proxy service. The *Relay* proxy service saves all result messages until it is assured that the recipient of the message, which is the mobile application, has received the messages. If after calling a service the connection of the mobile device to the proxy is lost, the mobile application, after establishing a new connection, queries the proxy to retrieve the result of the call to the remote service, instead of re-calling the remote service.

***DataTransfer* Proxy Service** The *DataTransfer* proxy service is used when the mobile application needs to transfer a large file to the remote service. This service is used to reduce the file transmission time, and to ensure no data loss happens during a file transmission in the presence of disconnection. The transferred file can be the executable of an offloaded task, an image that will be processed by the remote service, etc. The *DataTransfer* proxy service speeds up the file transmission by removing the need of re-transferring the file by the mobile application after the occurrence of a disconnection.

The larger the file to be transferred, the more likely there will be a disconnection between the mobile application and the remote service during the file transmission. If the remote service does not have any disconnection-aware mechanism that records how much data is already transferred from the mobile application, the mobile application has to re-retransmit the file from the start, after the re-connection. The time incurred by communication latency, and the

scarce wireless bandwidth are wasted for the re-transmission. The *DataTransfer* proxy service can be used to record the number of successfully transmitted bytes.

To use the *DataTransfer* proxy service the mobile application transfers the file to the *DataTransfer* proxy service and provides the *DataTransfer* proxy service with the contact information of the target machine that hosts the remote service. The *DataTransfer* proxy service receives the file and after being assured of the integrity of the transferred file, transfers it to the remote service.

If during the file transmission between the mobile application and the *DataTransfer* proxy service, a disconnection happens, the mobile application, after the re-connection, queries the *DataTransfer* Proxy Service to retrieve the number of already transferred bytes. By knowing this number, the mobile application does not transmit those bytes from the file. Essentially the mobile application transmits the remaining of the file to the *DataTransfer* proxy service.

The *DataTransfer* proxy service is implemented to be able to recognize a returning client, by checking its client identifier. This means that the *DataTransfer* proxy service is able to realize a client that wants to resume an interrupted file transmission, and is able to append the reminder of file to previously transferred bytes of the file.

***CodeExec* Proxy Service** The *CodeExec* proxy service is used for two purposes. First, the *CodeExec* proxy service executes the offloaded tasks from the mobile device to the proxy. A mobile application can offload the executable of a task to a proxy, and then provides the *CodeExec* proxy service with the input parameters needed to run the offloaded task. The *CodeExec* proxy service can also be used for moving a task from a remote location, such as a public cloud, to the proxy with the purpose of reducing the communication latency. The mobile application provides the *CodeExec* proxy service with location information of the task e.g., URL. The *CodeExec* proxy service then downloads the task to the proxy and executes it. In both cases, the *CodeExec* proxy service runs the tasks and returns the results to the mobile application.

If the executable file exists on the mobile device, the *CodeExec* proxy service can be used with the *DataTransfer* proxy service to offload the task to the proxy and run it on the proxy. In this way, the mobile application uses the *DataTransfer* proxy service to transfer the task file to the proxy, and then informs the *CodeExec* proxy service about the location of the task file on the proxy. This is used for the “remote print application”, described in section 3.3.1.3.

3.3.1.3 Remote Printing Application Using Proxy Services

The *Relay*, *DataTransfer*, and *CodeExec* proxy services can be used to implement the remote printing application. The *Relay* proxy service is used when the mobile application calls the

Printer Finder service. The *Printer Finder* service has a list of the addresses of printers in the campus. Upon receiving a request from a mobile application, the *Printer Finder* searches for a printer based on the preferences of the user. As described in section 3.3.1.1 receiving the response from *Printer Finder* may takes some time, especially if there is a large number of printers. The *Printer Finder* is a good candidate for being called through the *Relay* proxy service. The address of the *Printer Finder* service and input parameters are passed to the *Relay* proxy service. The *Relay* proxy service calls the *Printer Finder* service on behalf of the mobile application. The *Relay* proxy service saves the returned result in its database and sends the result to the mobile application (figure 3.3).

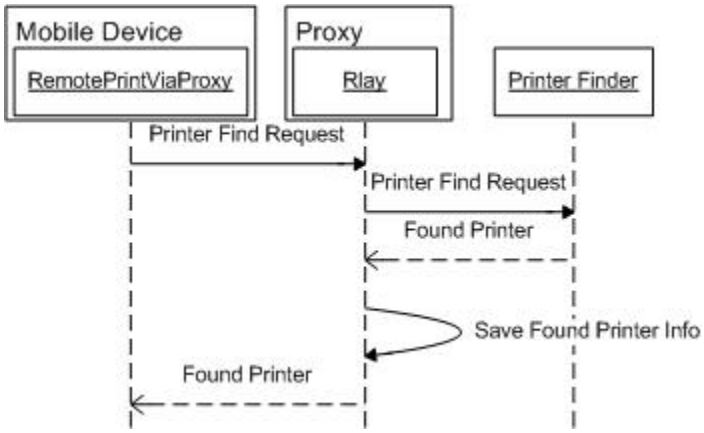


Figure 3.3: Call *Printer Finder* Via Proxy

The *DataTransfer* proxy service is used to transfer data from the mobile application to the proxy. For the remote printing application, the *DataTransfer* proxy service is used to transfer: (i) the document to be printed; and (ii) the *Printing Service Client* application, as an offloaded task, to the proxy. The *Printing Service Client* is used to communicate with the *Printing Service* and is provided by the university to users. The *CodeExec* proxy service is used by the proxy to execute the *Printing Service Client* application on the proxy. Figure 3.4 illustrates how a mobile application uses the *DataTransfer* and *CodeExec proxy services* to call the *Printing Service*.

3.3.2 Tourist Assistance Application

Examples of desirable augmented reality (AR) applications for mobile devices include e-Health [49], tourist assistance [26, 27, 61], gaming, navigation and industrial maintenance [53]. AR is defined as "a real-time direct or indirect view of the real world environment that has been enhanced/augmented by adding virtual computer generated information to it" [18]. AR applications help users gain better perception of the real world and enables users to interact with it.

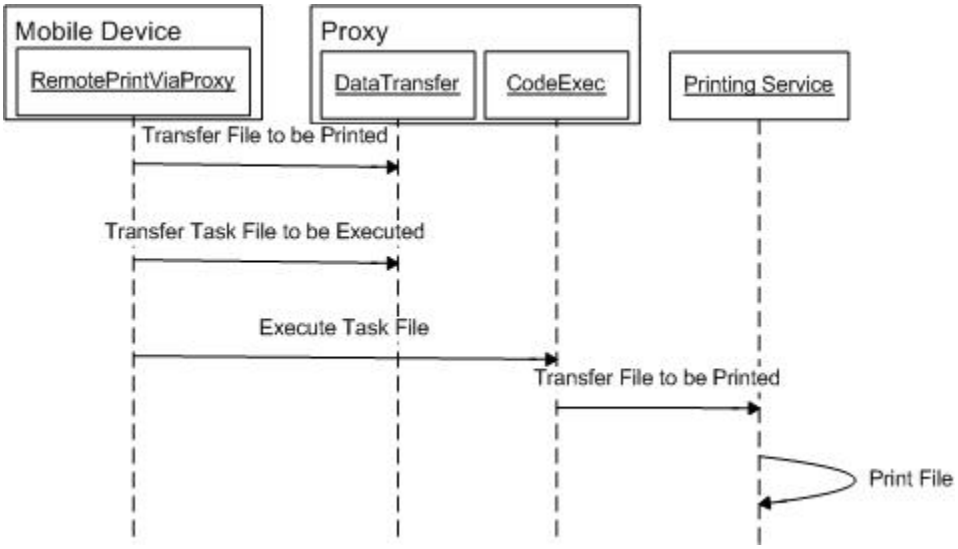


Figure 3.4: Call *Printing Service* Via Proxy

The information about the real environment is gathered using sensors and then provided to the AR application. The AR application processes the input information and augmented it with extra data. The extra data is then rendered to the users.

Huang et al. [33] notes that the AR paradigm has three main properties:

- combines the real world with the virtual world
- provides interactiveness with user in real-time
- provides 3-D objects to be added to the 3-D real environment

This means that AR applications often includes 3-D image processing tasks. Additionally, AR applications are expected to respond in real-time since they are interactive. Consequently, AR applications need computational power typically not found on mobile devices. In addition, the communication latency incurred by the limited bandwidth and the low reliability of wireless links, impacts the interactiveness of AR applications.

An AR application can be partitioned as follows. The mobile application on the mobile device gathers input data e.g., image, video, or voice. The data is transferred to a remote service for processing. This remote service is on a cloud. After processing the input data collected by the mobile application, the result is sent in an appropriate format to the mobile application, which is responsible for rendering the result to the user. It is shown that it is faster to offload the processing to a remote server rather than executing it locally on the mobile device, specially if the Wi-Fi connectivity is available [28].

As the second example application we consider an AR application that is designed for mobile devices and is used to assist tourists visiting historical sites. The application on the tourist's mobile device is used to capture images or video from the historical site to be sent to a remote service in the form of video or a stream of images. The remote service processes the stream of images or the video frames. The process might involve recognition of the site and extracting information about the site, such as historical information, from databases. The remote service sends the information in the form of text or retrieved images to the mobile application. The mobile application renders the received information in an appropriate format for the user. While the user is moving around in the site, the remote service can provides more detailed information about the view that the camera of the mobile device is pointing to.

3.3.2.1 Using the Proxy-based System

As described earlier, AR applications should have a real-time response. The communication delay when an AR application is installed on a remote location e.g., a public cloud, has negative impact on the responsiveness of these applications. One approach to reducing the impact of communication delay is to move the remote service, or only a part of it, to a location in the vicinity of the mobile device, whenever a mobile application needs the service. In this way the network latency between the mobile application and the service has less impact on the application.

The other difficulty of having this application on a mobile device is similar to the second problem introduced for the first example application: transferring data over wireless links from a mobile device to a remote server. The difference is that the tourist assistance application should transfer a stream of small files, as opposed to one large file, which are images, to the remote service. Although, the interruption in transferring a stream of files to the remote service would lead to the establishment of new connections to the remote service that might increase the response time of the service, since the service needs to repeat the view recognition process, which is time consuming. As a result, it is necessary to provide some mechanism that preserves the connection between the mobile application and the remote service, even in the presence of interruptions.

The issues mentioned above are not specific to this application and might affect other mobile applications with similar functionalities. The proxy services that can be used in this application to cope with these issues are introduced in section 3.3.2.2. Section 3.3.2.3 describes how the proxy-based system and introduced proxy services can be used to implement a mobile application for this example AR application that provides better service for the user.

3.3.2.2 Proxy Services Used for the Tourist Assistance Application

The *CodeExec* proxy service, as introduced in section 3.3.1.2, can be used to reduce the communication latency by downloading and installing the AR application, from a remote server, to the proxy and executing it. The URL of the downloaded service is provided for the mobile application and the mobile application can call the service on the proxy. Since the proxy is in the vicinity of the mobile application, the communication latency will be smaller.

***FileStreamer* Proxy Service.** The *FileStreamer* proxy service is used to transfer a stream of files, such as images, from the mobile device to the proxy. The *FileStreamer* proxy service handles disconnections, i.e. the *FileStreamer* proxy service is aware of the possible disconnections and is able to keep the connection to the remote service alive if mobile application is disconnected.

The *FileStreamer* proxy service is different from the *DataTransfer* proxy service in three ways: i) the *FileStreamer* proxy service is able to maintain the connectivity to the mobile application for transferring multiple files, while the *DateTransfer* proxy service is designed to transfer only one file, ii) Unlike the *DataTransfer* proxy service, the *FileStreamer* proxy service can begin delivering data, received from the mobile application, to the remote service, before finishing the data transmission from the mobile application. In this way, the output of the remote service can be presented to the mobile application at the same time that the mobile application is transferring the input data to the *FileStreamer* proxy service, iii) The integrity of transferred files are not guaranteed by the *FileStreamer* proxy service (unlike the *DataTransfer* proxy service) since it is assumed that the *FileStreamer* proxy service is used for applications that are time sensitive and a small number of corrupted files in the stream does not affect the performance of the remote service.

Similar to the *DataTransfer* proxy service, the functionality of the *FileStreamer* proxy service can be combined with the functionality of the *CodeExec* proxy service. The combination of these two proxy services, the *CodeExec* and the *FileStreamer*, enables software developers to develop mobile applications that gathers online data and feeds it to a remote service that process the data and provides the mobile application with the result of processing the data.

3.3.2.3 Implementing Tourist Assistance Application via Help of Proxy Services

When the mobile application on the mobile device, such as the tourist assistance application needs to transfer a large amount of data e.g., video or a stream of images, to the remote service, the network latency plays an important role in the performance of the application. The larger size of the data transferred between mobile application and the service exacerbates the

impact of network latency and may cause more number of disconnections. This condition has a negative impact on the application performance.

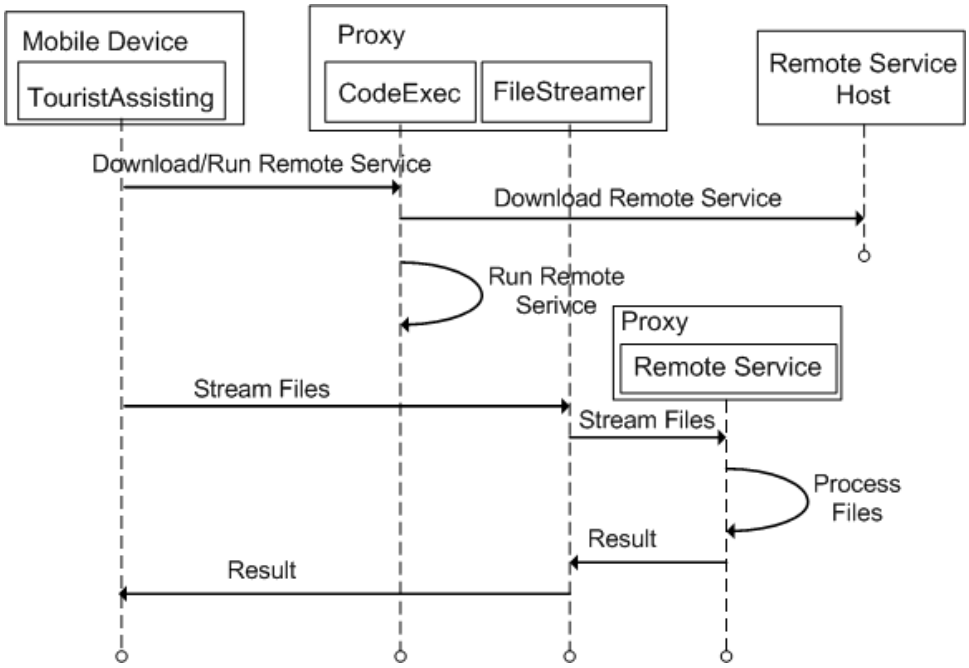


Figure 3.5: Run Tourist Assistance Application Via Proxy

Using our proxy-based infrastructure, the tourist assistance application can connect to a nearby proxy and ask the proxy to download and install the remote service. Afterwards, the mobile application can call the service on the proxy machine which is located in a nearby place and, as a result, the network latency will be kept lower. The tourist assistance application uses the *CodeExec* to download the required service from a remote host to the nearby proxy and run it and subsequently reducing the latency. The *FileStreamer* proxy service is used to transfer the stream of image files from the mobile application to the proxy. The *FileSteamer* proxy service provides the disconnection handling, i.e. *the FileStreamer* proxy service is aware of the possible disconnections and is able to re-connect and continue the communication between the mobile application and the service without the intervention of the mobile application or the service.

The combination of these two proxy services, the *CodeExec* and the *FileStreamer*, enables programmers to create mobile applications for other AR services that need the transmission of an extensive amount of input data in the form of stream of images (or other formats) from the mobile application and also requires extensive amount of processing on the input data.

3.4 Analysis

A comparison of the application models in various mobile cloud computing systems is provided by Kovachev et al. [46]. The application models mentioned in Kovachev et al. [46] are compared based on several aspects. Some of the aspects, related to the proxy-based mobile computing environment, are used here to compare the programming models introduced in some of the work presented in section 2 and also to describe the characteristics of the proposed programming model. The aspects used for the comparison of programming models are the following:

- Dynamic vs. static partitioning
- Solution generality
- Programming abstraction
- Compatibility with old systems
- Application mobility

The analysis related to the programming abstraction is provided in the related work section of chapter 6.

3.4.1 Dynamic vs. Static partitioning

The most common use of proxies is to execute tasks on behalf of mobile devices. As described previously in section 2.1, in these systems the computationally heavy parts of an application is offloaded to a proxy. Some of the work presented in section 2.1 do not involve any partitioning. Instead the entire task is offloaded to the proxy. While in some other work, the task is decomposed into several partitions and the computationally heavy partitions are offloaded from the mobile device. In these systems there is a need to find the best partitioning of the application to achieve a specific goal, such as decreasing the communication cost or the resource usage on the mobile application.

The work that involves application partitioning can be divided into two categories: dynamic partitioning and static partitioning. With static partitioning, at the start of the execution of the task, the task is partitioned, based on specific performance factors and some partitions are offloaded, while the remaining partitions are executed on the mobile device. This partition distribution between the mobile device and the proxy does not change during the task execution. In dynamic partitioning, factors used for partitioning are monitored constantly and if

needed the partition distribution changes. This means that during the task execution a partition might be migrated from mobile device to the proxy or vice verse.

In section 2 some of the work presented uses static partitioning. This includes Cuckoo [40], Chroma [15], *Economic Mobile Cloud Computing* [48], DYNAMO [51], and CloneCloud [21]. Additionally, work including dynamic partitioning are presented in section 2 as well, i.e. MAUI [23], Elastic HTML5 [67], and *Partial Offload to Cloudlet* [66].

In our system, we employed a static partitioning approach, which is similar to the client-server model. We assume that computational heavy part of the application is implemented in the form of services and it is installed on the proxy or other powerful remote resources. A light weight application should be provided to be run on the mobile device. The communication cost between the mobile application and the remote server is controlled by employing an efficient proxy discovery approach, which is described in chapter 4. In addition, by adopting the static partitioning approach, the application development is simplified for developers. The monitoring of environment is carried out by the system, although, not for the sake of application partitioning. As described in the chapter 5, monitoring is done for handoff between proxies.

Although our work primarily focuses on static partitioning, the proposed proxy-based system supports dynamic partitioning by allowing a mobile application to offload computational tasks to a proxy. However, details related to determining the dynamic partitioning is not considered within the scope of this work and is complementary to our work.

3.4.2 Solution generality

The solution generality is an attribute of the programming model that shows how much a model can be used to provide a solution for different problems. Some approaches are able to provide a solution for specific problems, while some approaches are able, or have the potential to be used for a wider range of problems.

Among the approaches introduced in section 2, the *i5 Multimedia Cloud Architecture* [44] and the *Proximal Workspace* [63] are examples of work that provide higher solution generality.

The *i5 Multimedia Cloud Architecture*, introduced in section 2.1.2.3, uses a multi-layer architecture to support the multimedia mobile applications. New multimedia services, upon creation, can be added to the multimedia services layer.

The *proximal workspace* proposed by Taylor et al. [63] (section 2.2.7) provides a set of middleware utilities for specific AR tasks, such as rendering, sensor input processing, and object tracking. The authors specified that although it is not expected to run arbitrary code on the *proximal workspace*, the utilities provided on the *proximal workspace* are designed to offer common required tasks for AR applications. These utilities are designed to be reusable

by various applications.

As described earlier in section 3.2.2, the proposed infrastructure is able to support various applications with different requirements. Common requirements of mobile applications, such as transcoding and buffering, can be implemented as public *proxy service* that are accessible by service providers, so that service providers can offer their services to mobile users. Service providers are also able to provide their own private proxy services to customize their services for their mobile clients, or because of security concerns.

In addition, the proposed infrastructure can be used for services in various areas, as opposed to the *i5 Multimedia Cloud Architecture* [44], which is designed for supporting multimedia applications, and the *Proximal Workspace* [63], which provides services for AR applications.

3.4.3 Compatibility with old systems

There are many services already implemented that target applications that execute on powerful desktop machines. It is useful if mobile computing infrastructures provide required facilities for mobile clients to access and use these services, without changing (or minimizing changes) to the remote service. The proposed mobile computing infrastructures, introduced in chapter 2, require the redesign and development of remote services that are compatible with the proposed infrastructure. Although, the work presented in Taylor et al. [63] is an example of mobile computing solution that is compatible with an old system.

In Taylor et al. [63] (section 2.2.7) the remote service is the Google Earth Ancient Rome 3D. The client application sends the input data, including the location, to the service and the Google service returns a large amounts of multimedia data that describes the area. The client application should be able to render the returned data from the remote service quickly to the user and also be able to save large amount of data. The minimum requirements for the machine hosting the client application and the network that the client application is connected to might not be met when the client is running on a small mobile device, such as a smart phone. Taylor et al. [63] propose a *proximal workspace* that is located close to the mobile device and runs the client application of the remote service. The client application on the *proximal workspace* intercepts the data sent from the mobile device and the remote service. In this way the problem of resource-poverty of mobile device and network latency is solved, while no alteration is made to the remote service.

3.4.4 Application mobility

In the mobile computing infrastructure there is sometimes a need to migrate the application from one device to another device. The migration usually occurs from the mobile device to the

proxy. It is desirable to provide the live migration facility, i.e. migrating an application between two hosts while the application is running. It is important to do the live migration transparent from the user's perspective, while incurring the least amount of distraction to the functionality of the application. Two approaches are mostly employed for the application mobility in the work presented in chapter 2:

- Using VM technology
- Multiple code implementations

The VM technology is used in *CloneCloud* [21] and is proposed in Satyanarayanan et al. [60]. In these approaches VM migration technology is used to provide application mobility. The work of Satyanarayanan et al. involves VM overlay migration. However, it is assumed that a copy of the mobile device VM is already running on the cloudlet. The drawback of using VM migration is that it is time consuming and generates large traffic load on the network. VM migration between different platforms even takes longer [46].

The second approach i.e., using multiple code implementations, assumes that the code that has the same functionality of the code on the mobile device and already is available on the proxy. This means that two versions of an application should be implemented: one to be executed on the mobile device and one to be executed on the proxy. An example of this approach is seen in Fesehaye et al. [25] (section 2.2.3). In Verbelen et al. [66] (section 2.2.4), since OSGi components should be executed on both mobile devices and cloud nodes, the native libraries are compiled for both x86 based machines with Linux and ARM-based mobile devices with Android. At the run time, the appropriate compile library is loaded. In *MAUI* [23] (section 2.1.1.1) two versions of migratable methods should be provided: one for local execution on the phone and the other for the remote execution. The *Cuckoo* framework helps the developer to bundle two versions of code in one package, and at the runtime depending on the location of execution the appropriate version is used. Although, it is again the responsibility of the application developer to provide two versions. This approach causes additional burden for the developer that intends to develop an application that uses the system.

In this work the task migration happens during the handoff operation. The proxy services can be migrated to another proxy during execution. It assumed that the code of the migrated proxy services is already available on the target proxy. In this approach the state of the migrated proxy service to the target proxy may need to be transferred, as described in section 5.2. This approach in comparison to the approach that uses the VM technology is faster and generates less network traffic. In addition, in comparison to approaches that require multiple implementation, less programming effort is needed.

3.5 Summary

This chapter discusses the benefits of using proxies. The literature review in chapter 2 provided a holistic view of the state-of-the-art work done in this area. The literature reviewed provided insight into realizing the common uses of proxies proposed. This insight helped us in designing the proposed proxy-based architecture presented in this chapter.

The proposed proxy-based system takes the advantage of having multiple proxies. It helps to provide less communication delay for delay-sensitive applications, load balancing between proxy, as well as other useful services.

The components of the system, their responsibilities, and the interactions between them presented in this chapter. Two example scenarios are provided to show how real-world problems can be solved using the proposed system.

The services and provided APIs are discussed in Chapter 6.

Chapter 4

Proxy Discovery

The proposed proxy-based infrastructure involves the use of multiple proxies. As described earlier, the presence of multiple proxies has several advantages:

- Having multiple proxies provides a mechanism for balancing the load incurred by mobile applications between proxies.
- Not all proxies are able to provide service for all mobile applications due to not having required resources.

A proxy discovery mechanism is needed by mobile applications to discover the proxy that best suits the needs of the mobile application that requires the use of a proxy.

One approach is to provide mobile applications with the information of all available proxies and have the mobile application responsible for discovering an appropriate proxy, based on the requirements of the mobile application and resources available on proxies. With this approach each mobile application has the code for a proxy discovery mechanism. In this approach the mobile application is not dependent on any other component for discovering a proxy. However, this approach has several disadvantages:

1. If there are a large number of proxies, the process of finding the best proxy might need computing resources that are not available on the mobile device. The delay incurred while selecting a proxy may not be acceptable to the user of the mobile application.
2. It is difficult to collect and update information about proxies. After adding a new proxy to the system, or updating the information of a proxy, there is a need to notify all mobile applications of the changes. This approach is not scalable when there is a large number of mobile applications. A possible approach to this problem is to have the information of all proxies in an online repository. This online repository is made accessible for all mobile

applications. The problem with this approach is wireless links have limited bandwidth. Large sizes of this repository may negatively impact mobile application response time.

In this work we propose the use of multiple *PFSs*. A mobile application contacts a *PFS* to discover a proxy that best suits its needs. Each *PFS* has access to a local repository of proxy information. A proxy registers itself at one or more *PFSs* and each contacted *PFS* places proxy information in its repository. If the information of a proxy changes, the proxy informs the *PFSs* that it is registered with. Since proxies might register at different sets of *PFSs*, it is possible that the list of proxies registered at *PFSs* are different.

To find a proxy, a mobile application contacts at least one *PFS*. It can speed up the proxy discovery operation since the mobile application has the choice of accepting the first proxy found by one of the *PFSs*. Contacting more than one *PFS* can also increase the reliability of the proxy discovery process. If a *PFS* was not able to discover an appropriate proxy, other *PFSs* might be able to discover one. The mobile application can also compare the proxies discovered by multiple *PFSs* and select the proxy that best meets the needs of the mobile application. Contacting multiple *PFSs* has several disadvantages: (1) It increases the traffic load on the network. (2) More data is transferred/received from/to the mobile device, which results in higher battery consumption. On the other hand, if the mobile application limits itself to contacting only one *PFS* the mobile application might not get the proxy that best suits its needs.

Having multiple *PFSs* removes the problem of a single point of failure. If a *PFS* is not accessible, there are other *PFSs* to be contacted. Thus, mobile applications should be provided with the access information of multiple *PFSs*.

In this chapter the protocol that enables mobile applications to discover the proxy that best suits the needs of the mobile application is presented. Section 4.2 describes the proxy discovery protocol which includes information provided by a mobile application to a *PFS*, the use of proxy attributes to select a proxy, and the interactions between the mobile application, *PFS* and proxy. Section 4.4 discusses our evaluation of the architecture and protocol.

Before focusing on our proposed approach, a review of the methods, that other proxy-based work employed to provide the proxy address for mobile applications, is presented.

4.1 Related Work

This section presents a review of related work related to service and proxy discovery and identifies gaps in this work.

4.1.1 Service Discovery Protocols

One can think of a proxy as a service and thus proxy discovery can be thought of as service discovery. With the increase in the number of devices connected to the Internet and the services available on-line, the need for service discovery methods emerged. Several service discovery techniques including the Service Location Protocol (SLP) [11], Universal Plug-and-Play (UPnP) [12], Salutation [10], Jini [5], DEAPspace [52], etc. are introduced. These techniques focus on providing service description, search algorithm, query format and query matching approaches.

Some of these techniques, such as SLP, Jini, and Salutation, employ centralized architectures, while others, like UPnP and DEAPspace propose to use a distributed peer-to-peer structures. In the centralized methods, one or multiple nodes are used as directory servers that save the information about registered services. If a client needs to find a service, it should send the query to the directory server. In the peer-to-peer approaches each device can provide services and at the same time use the services provided by other devices. The peer-to-peer approaches do not scale very well because of the large number of messages generated. For example, the UPnP can be used among devices connected to the same LAN and the DEAPspace is designed to be used in one-hop wireless networks.

A problem with using existing service discovery protocols is the assumption that discovery protocols are often assumed to provide information within a local area network e.g., UPnP and DEAPspace, or within an enterprise e.g., SLP and Jini. To increase flexibility with respect to associating mobile devices with applications, we do not want to assume that the proxies and mobile devices must be in the same network.

A second problem is the assumption that attributes used to describe a service are static. With static attributes the value of the attributes are assigned once when the service registers. The protocols do not provide facilities to monitor attributes whose value may change over time.

4.1.2 Proxy Discovery in Other Work

Most of the work presented in chapter 3 does not provide an approach for discovering proxies. This section briefly describes the little work that does propose proxy discovery mechanisms.

The Cuckoo framework, Kemp et al. [40], introduced in section 2.1.1.4, is an example of a system that enables the offloading of computation from the mobile device. In this system it is assumed that there may be multiple available remote resource providers that accept the execution of the offloaded task. These remote resources are actually proxies that are used for the task offload. The *Resource Manager* component on the mobile device, maintains the information about the registered remote resources. If the remote resource has an accessible

display, the information about the resource, that is encoded in a QR code, is shown on the display. The camera available on the mobile device is used to scan the QR code to retrieve information about the remote resource. If the display of the remote resource is not accessible, the information about remote resource should be transferred from the remote resource to the mobile device in the form of a description file. As described, the registration of the remote resources at the *Resource Manager* is manual and no automation is provided.

The work proposed by Rajachandrasekar et al. [56], presented in section 2.1.1.6, includes a node called proxy that receives parameters for running a multi-threaded task from mobile applications and distributes the threads among nodes of a cluster. Before distributing the threads, the proxy finds the best nodes in the cluster. The best nodes are defined as those that have the most computational capacity and have the smallest distance to the proxy. The distance is measured using the RTT between the proxy and the cluster node. Nodes are sorted based on their computational capacity. If two nodes have the same capacity, the one with the smaller distance is chosen. In finding the best nodes the condition of the nodes, i.e. the CPU load of the nodes, is not taken into consideration.

The work presented by Trung et al. [64], introduced in section 2.1.1.10, considers the cost and resource availability of proxies for offloading computation. The work uses a primary Replication Gateway (RG) to submit replicas of one task, submitted from the mobile device, to multiple RGs. The access information of the primary RG should be available to the mobile application beforehand. It is the responsibility of the primary RG to find other RPs based on the cost and the resource availability of the RPs. The primary RP, using an algorithm, finds the optimal number of RPs, based on the cost of replication creation on each RP, that fits the budget of the mobile user. After having the optimal number of replicas, the RPs are sorted in descending order of replication creation cost. The first RP with sufficient resources is chosen.

The work presented in section 2.1.1.8, proposed by Guan et al. [30], uses a proxy to discover remote resources. The mobile device needs to first find a proxy. Finding the proxy assumes that the mobile device is close enough to the proxy machine so that the mobile device's *Device Proxy Module*, is able to find the proxy machine using a local service discovery protocol.

The work described in section 2.2 focuses on reducing latency by using a nearby proxy. However most of the work presented in that section assumes the proxy exists in the proximity of the mobile device, and the mobile device knows the access information of the proxy beforehand. The work proposed by Bahl et al. [14] uses cloudlets installed on the "wireless access network", WiFi hotspots, and peer mobile devices. In the "Pocket Cloudlet" [45] the mobile device itself is the cloudlet that is used as the storage. As a result, no cloudlet discovery technique is needed.

Satyanarayanan et al. [60] assumes that the proxies can be installed in public places such as coffee shops or doctor offices and proposes mechanisms for providing security, but there is no discussion on who is the provider of proxies and how the access information of the proxies can be made available for the mobile applications.

The work proposed by Verbelen et al. [65, 66] was described in section 2.2.4. The cloudlet proposed in this work is made of multiple devices in the vicinity of each other. A *Cloud Agent* (CA) on one of these devices on the installation of the components of an application on the cloudlet nodes. A decision algorithm introduced in Verbelen et al. [65] uses the information about the available bandwidth between cloudlet nodes, the CPU speed of nodes and the latency of the links, along with the requirements of the application components in terms of execution time threshold, CPU cycles and bandwidth needs, tries to find an optimal placement for the components on the cloudlet nodes. The problem with this approach is that the information about the the CPU, bandwidth and latency required by every application component should be provided by the application developer.

Two proxy-based mobile computing systems are introduced in the related work section of chapter 5. These systems include handoff mechanisms between proxies that is invoked when the network condition. Samimi et al. [59] provides a mechanism to find the best proxy based on the communication links condition. However in the other work, Bellavista et al. [17], only mentions that the buffering proxy should be close to the mobile device.

The work proposed by Samimi et al. [59], described in section 5.1.1, uses two proxies to support the applications on mobile devices: primary proxy and transient proxy. The transient proxy is in the proximity of mobile device and may change with the movement of mobile device. The primary proxy is chosen from nodes on an overlay network. The factors for choosing the primary proxy are the security policies of the client and the service provider, the RTT between the overlay node and communication endpoints, i.e. mobile device and the service, and the computational load on the overlay node. No factors are mentioned for choosing the transient proxy. The transient proxy should be a node on the network that the mobile device is connected to.

4.1.3 Gap Analysis

Very little of the work is concerned with mechanisms for making available the access information of a proxy, or choosing the most appropriate proxy among several available proxies.

A problem with using existing service discovery protocols and some of the work presented in sections 4.1.1 and 4.1.2, e.g. Rajachandrasekar et al. [56], is the assumption that most attributes used to characterize a service are static. With static attributes the value of the attributes

are assigned only once when the proxy registers. An example of a static attribute for a proxy is CPU speed. Although service discovery protocols allow for the specification of dynamic attributes e.g., proxy CPU load, the protocols do not provide facilities for monitoring dynamic attributes.

The other problem with the approaches described in this section in finding a proxy is that they provide information within a local area network or within an organization. Proxies are assumed to be accessed through WiFi even though mobile applications can also use cellular technologies. To increase the flexibility with respect to associating mobile devices with applications, we do not want to assume that the proxies and mobile devices must be in the same network or that only WiFi can be used (or only cellular).

Scalability is the missing aspect in several of the proxy finding approaches presented in this section. Some of the work assumes that the mobile application can save information of all proxies e.g., Kemp et al. [40], and some work assumes the presence of only one proxy to be used by all mobile applications e.g., [64, 65, 66]. These approaches are not applicable when the the number of mobile applications increases, or the number of proxies grows.

Proxy discovery mechanisms proposed for mobile computing environment need to be scalable, and be aware of the changing conditions of the environment. Additionally, the proposed approach should not limit the mobility of the mobile devices.

4.2 Proxy Discovery Mechanism

The proxy discovery mechanism is presented in this section. This mechanism includes the registration of proxies at *PFSs*, the handling of discovery requests from mobile applications to the *PFS*, and the selection algorithm used by a *PFS* to find an appropriate proxy for the requesting mobile application.

4.2.1 Proxy Registration

When a proxy first comes on-line, it registers itself with one or more *PFSs*. The information provided by the proxy at registration should include all the information that is needed by the *PFS* to find the best proxy for a mobile application. The information includes:

- The list of proxy services available on the proxy.
- The cost model that the proxy uses to charge its associated mobile applications

- The privacy constraints of proxy services that specify the accessibility of the proxy services for mobile applications. A proxy service can be either publicly available, or accessible through authentication.

Upon receiving a registration request from a proxy, the *PFS* can access other resources to gather more information about the proxy. For example, the *PFS* might contact a trust system to obtain the information about the reputation of the proxy.

The information about proxies saved at *PFSs* can be updated later by proxies or by other resources that the *PFS* uses to gather information about proxies. For example, if the list of available proxy services at a proxy changes, the proxy informs the *PFSs* at which it is registered. As another example, if the trustworthiness of a proxy changes, the trust system contacts the *PFS* to update the trust information of proxy.

4.2.2 Proxy Discovery

The process of proxy discovery is depicted in figure 4.1. In this figure the interaction between mobile application, *PFS*, and proxies is graphically depicted. In this figure it is assumed that the mobile application contacts only one *PFS*.

Step 1 - Starting the Process

A mobile application invokes the proxy discovery operation by sending a request to the *PFS*. The request sent to the *PFS* should include the information that a *PFS* needs to find a suitable proxy. The information might include the IP address of the mobile device, which is needed to find the network latency between the mobile device and the proxy or the remote service, the list of required proxy services needed by the mobile application, the information of any remote services that will be accessed by the mobile application, such as the URL of the remote service, etc.

The preferences of the mobile application, that will be taken into account by the *PFS* for proxy discovery, should be included in the request. In the preferences the mobile application indicates the weight of attributes for choosing a proxy that best suites the needs of the mobile application. Preferences take the form of pairs where the first element of a pair is an attribute and the second is the weight. For example, the mobile application declares in the request, the network delay from mobile device to the proxy, the network delay between the proxy and the remote service, the acceptable cost of using the proxy, and the acceptable reputation level of the proxy as attributes, and assigns a weight to each attribute. The sum of weights should be equal to one.

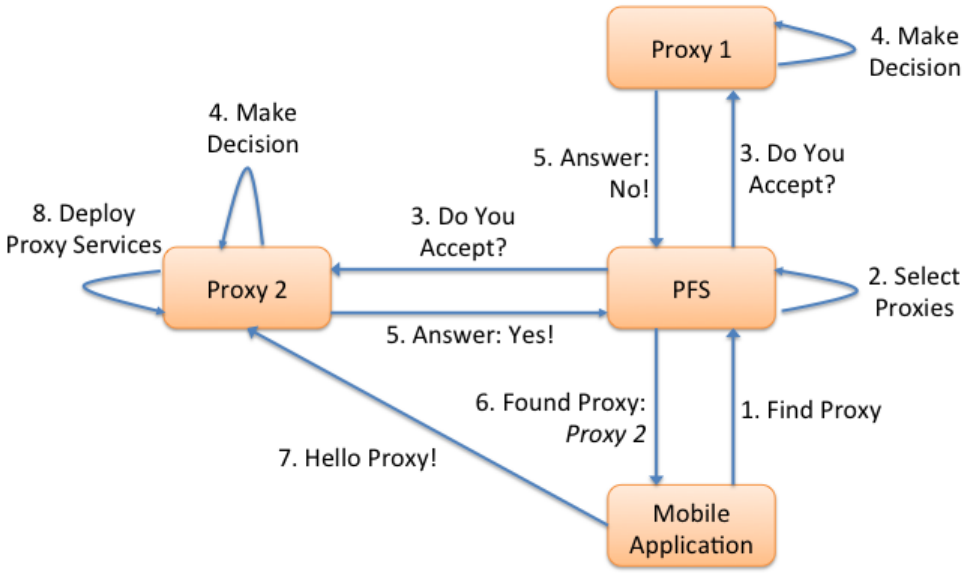


Figure 4.1: Proxy Discovery

To illustrate the use of weights consider a mobile application that needs to call an interactive remote service. An instance of the remote service is started on the proxy. Since this is an interactive application, there is a preference for a proxy with a low network latency between the mobile device and the proxy. This causes the mobile application to assign a higher weight to network latency. The mobile application may wish to use the service as quickly as possible and hence it prefers a proxy that does not need to download and install the missing proxy services. A weight may be assigned to the availability of all required proxy services.

Step 2 - Choosing a List of Proxies

Upon receiving the request for a mobile application, the *PFS* determines a ranked list of proxies. A rank is assigned to a proxy based on user preferences. The algorithm used by the *PFS* to determine a ranked list of proxies is presented in more detail in section 4.2.3.

Step 3 - Contacting Discovered Proxies

Before selecting a proxy from the list created in step 2 to the mobile application, there is a need to inquire the proxy about its ability to accept the mobile application as a new client. Since a

proxy may reject a request from the mobile application, the *PFS* makes an inquiry to all of the proxies in the ranked list. Thus if a higher ranked proxy rejects a request, a lower ranked proxy may accept and that proxy would be used.

The *PFS* could employ various strategies in contacting the chosen proxies:

- The PFS contacts the proxies one by one from the beginning of the list. The *PFS* does not proceed to next proxy unless it receives a denial of its request from the most recently contacted proxy.
- The PFS simultaneously contacts all proxies in the list, and waits to receive responses from all proxies. The proxy with the highest rank that responded positively to the request is chosen. This approach can generate lots of network traffic, especially if there are a large number of proxies, but can be faster than the first approach
- The PFS simultaneously contacts all proxies, and chooses the first proxy that accepts the request. This approach is the fastest, although the selected proxy may not best suit the needs of the mobile application.

In our work we use the second approach which is relatively fast and since we are not including a large number of proxies in our experiments this approach does not generate a large amount of load. In the example depicted in figure 4.1 two proxies, proxy 1 and proxy 2, are chosen as potential proxies that conform with the preferences of the mobile application. The *PFS* sends the request of the mobile application to these proxies.

Step 4 - Making the Decision to Accept a Mobile Application Request by Proxies

The proxies that receive the request from the *PFS*, decide whether to accept or reject the mobile application as a new client. A proxy makes this decision based on its resource availability e.g., CPU load, memory usage, number of network connections, the available proxy services, and its ability to provide the required proxy services that are not available. The proxy, considering the kind and amount of resources required by the requesting mobile application, should decide if it has sufficient resources to support the mobile application.

A proxy may reject a request if the CPU load on the proxy is already high and the requesting mobile application needs to use the *CodeExec* proxy service for the purpose of offloading the execution of a task to the proxy. In this case accepting the request of this mobile application may result in exceeding the CPU load threshold value. As another example, if the proxy does not have all the required proxy services available, and cannot download the missing required proxy services, the proxy should reject the request. A proxy might not be able to download

the missing proxy services if they do not exist in on-line repositories accessible by the proxy, or the proxy does not have enough available computing resources to download and install the missing proxy services.

The conditions for accepting or rejecting request can be stored as a set of policies at the proxy.

Step 5 - Replying to the Inquiry from the *PFS*

After making the decision about the request from the *PFS*, the proxy informs the *PFS* about its decision. In the example shown in the figure 4.1, proxy 1 has decided to reject the request, while proxy 2 has accepted the request.

Step 6 - Proxy Selection

The *PFS* selects only one proxy from the set of proxies that responded positively to the *PFS* in step 5. If all the contacted proxies decline the request, the proxy finding request of the mobile application is dropped.

Upon selecting a proxy, the *PFS* informs the mobile application about the information of the accepting proxy, including the IP address of proxy and the URI's of the proxy services on the proxy that mobile application intends to call, etc.

The acceptance message sent by the accepting proxy contains a client identifier issued by the proxy for the mobile application. The issued client identifier is unique throughout the system and is used by the proxy and proxy services to keep the state information related to the requesting mobile application.

Step 7 - Contacting the Best Proxy

The mobile application, after receiving the information of the chosen proxy, contacts the proxy by sending a *Hello Proxy* message that includes the client identifier issued by the proxy. Other information that may be included in the *Hello Proxy* message by the mobile application. For example the attributes and acceptable thresholds of attributes, e.g., the RTT between mobile device and proxy, that should be provided for the mobile application. This information is used to decide a proxy that best suites the needs of the mobile application.

Step 8 - Preparing Required Proxy Services

The proxy uses the client identifier included in the *Hello Proxy* message to retrieve the request of the mobile application. After retrieving the request, the proxy prepares the required proxy

services. The preparation includes the downloading of missing proxy services from an online repository of proxy services or downloading them from a local repository. This is followed by installation. After the successful installation of all proxy services, the proxy sends an acknowledgment message in response to the *Hello Proxy* message. After receiving the acknowledgment from the proxy, the mobile application knows that the proxy services on the proxy are ready. As a result, the mobile application can start to access proxy services when needed.

4.2.3 *PFS* Algorithm for Selecting Proxies

The algorithms used by the PFS for selecting proxies is described in this section. Proxy information and the preferences of mobile application are given to algorithm as input.

4.2.3.1 Attributes

A set of attributes is used by the *PFS* to find a list of proxies for a mobile application. We categorize attributes into two categories. The first category includes static attributes whose values do not change or change infrequently. An example of a static attribute is RAM. The second category consists of dynamic attributes that may frequently change. Examples of dynamic attributes are the network latency and the processing load on the proxy. The network latency depends on the network condition and may change frequently. Likewise, the load on the proxy depends on the applications running on the proxy and might change frequently.

The values of static attributes can be gathered during the registration phase of the proxy at the *PFS*. The values of static attributes are either provided by the proxy, or are queried from relevant sources. For example, the physical characteristics of proxy machine e.g., RAM and CPU speed can be included in the registration request sent by the proxy. However, the cost model that the provider of proxy uses to charge clients can be queried from the agent of proxy provider.

The value of an attribute might change during time. As a result, the *PFS* should provide a mechanism that allows for updating of attribute values. For example, if the list of available proxy services on a proxy changes, the proxy will be able to contact the *PFS* and update this information. The *PFS* may periodically contact other sources of information to receive updated values for attributes. For example, the *PFS* can contact a third party trust system periodically, to receive updated information on the trustworthiness of proxies.

Since values of attributes in the second category e.g., dynamic attributes, change more often, the *PFS* needs to frequently update attribute values. Some attributes may need to be updated each time a proxy is to be selected. For example, the *PFS* needs to contact proxies to find the current network latency or the current processing load on the proxy. An alternative ap-

proach is to have proxies and other resources periodically send the values of attributes to *PFSs*. This allows a *PFS* to select a proxy without making a request for information from a proxy and other resources, since the *PFS* already has the required information to make decisions. The problem with this approach is that there may be a large number of proxies registered with a *PFS* which would require the *PFS* to handle many periodic messages from proxies and other resources. This load could increase the response time of the *PFS*.

4.2.3.2 Preferences of Mobile Application

The *PFS* also considers the preferences of the mobile application. The preferences of the mobile application are indicated in the proxy find request that the mobile application sends to the *PFS* in the form of a set of attributes. The mobile application also includes the importance of each attribute by assigning a weight to each attribute in the request. The attributes that are of the interest of the mobile application can be a subset of the attributes that are known by the *PFS*.

4.2.3.3 The Algorithm

Proxy selection is described in algorithm 1. The input includes proxy information and the preferences of the mobile application.

The set of proxy information is represented by the set Proxies:

$$\texttt{Proxies}=\{p_1,p_2,\ldots,p_n\}$$

where p_i represents the i_{th} proxy:

$$p_i = \{(a_1,v_1),(a_2,v_2),\ldots,(a_m,v_m)\}$$

a_j represents an attribute type and the value of that attribute is denoted by v_j.

The preferences of a mobile application can be represented as follows:

$$\texttt{ClientPreferences} = \{(a_1,w_1),(a_2,w_2),\ldots,(a_m,w_m)\}$$

In this representation, each pair (a_i, w_i) shows an attribute, i.e. a_i, and its weight assigned by the mobile application, i.e. w_i.

Lines 1-7 of algorithm 1 represents the updating of dynamic attributes. After updating the values of dynamic attributes for all registered proxies, the *PFS* calculates the rank of each proxy. The attribute values must be normalized before ranking the proxies. Normalization of the values is needed since the range of values of each attribute is different and to calculate a weighted average of the attributes there is a need to map all attributes to the same range e.g., 0 to 100. Normalizing is done through the *NormalizeAttributes* function in line 8 of the algorithm. The *NormalizeAttributes* function is shown in algorithm 2.

Input: Proxies, ClientPreferences
Output: K Best Proxies
1 **for** *i from 1 to n* **do**
2 **for** *j from 1 to m* **do**
3 **if** a_j *is dynamic* **then**
4 $p_i.v_j$ = UpdateValue(p_i, a_j);
5 **end**
6 **end**
7 **end**
8 NormalizedProxies = NormalizeAttributes(Proxies);
9 **for** *i from 1 to n* **do**
10 $np_i.rank = \sum_{j=1}^{m} w_j * np_i.v_j$;
11 **end**
12 KBestProxies = FindKProxiesWithLargestRank(NormalizedProxies);

Algorithm 1: Proxy Selection

To normalize values of an attribute, the algorithm finds the worst and best values of the attribute among all proxies. For some attributes the best value is the highest value, e.g. the percentage of available proxy services, while for some other attributes the best value is the lowest value e.g., the RTT. If the best value is mapped to 100 and the worst value is mapped to 0, then the values of attribute for all proxies is mapped to the normalized value, which is between 0 and 100.

Discovering the best and worst values is carried out in lines 1-4 of the algorithm 2 where a_{best} and a_{worst} represent the best and worst values of attribute a among all proxies, respectively.

For each proxy, a copy of the proxy is assigned to a temporary variable, *tempProxy*, in line 6 and the calculated normalized values of a proxy are saved in *tempProxy*. The *tempProxy* variable is used to keep the original values of the proxy intact, since in another proxy discovery process, the original values are needed again. The task of standardizing the values of all proxies is seen in lines 7 - 9.

After saving the normalized values of all attributes of a proxy in *tempProxy*, the *tempProxy* variable is added to the *NormalizedProxies* set in line 10. After repeating this process for all proxies, the *NormalizedProxies* set is returned to algorithm 1.

By having the *NormalizedProxies* and the *ClientPreferences*, the weighted average values of attributes for each proxy can be calculated. This calculation is done in lines 9 - 11 of algorithm 1. The weights used in line 10 of algorithm 1, to calculate the rank of a proxy, are taken from the *ClientPreferences*, which is one the inputs to the algorithm. The calculated *rank* represents the appropriateness of the proxy. A larger *rank* value means a more appropriate proxy for the requesting mobile application. Finally, the K proxies that have highest calculated

Input: *Proxies*
Output: *NormalizedProxies*
1 **for** *each j from 1 to m* **do**
2 | $a_{j_{best}}$ = *FindBestValueOfAttribute*(a_j, *Proxies*);
3 | $a_{j_{worst}}$ = *FindWorstValueOfAttribute*(a_j, *Proxies*);
4 **end**
5 **for** *each i from 1 to n* **do**
6 | *tempProxy* = p_i;
7 | **for** *each j from 1 to m* **do**
8 | | $tempProxy.v_j = \frac{p_i.v_j - a_{j_{worst}}}{a_{j_{best}} - a_{j_{worst}}}$;
9 | **end**
10 | *NormalizedProxies.add(tempProxy)*;
11 **end**
12 **return** *NormalizedProxies*;

Algorithm 2: Normalize Attributes Procedure

rank are found in line 12 of algorithm 1.

4.3 Prototype of the Proposed System

Three attributes are considered in the proxy discovery mechanism of the implemented prototype of the proposed system:

- The percentage of used CPU capacity on the proxy
- The percentage of available required proxy services on the proxy;
- The sum of the RTT value between the proxy and the mobile device, and the RTT value between the proxy and the remote service that is to be used by the mobile application

The percentage of used CPU on the proxy is important for mobile applications that need to offload CPU-intensive tasks to the proxy.

If the mobile application functionality does not involve any call to a remote service, only the RTT between the mobile device and the proxy will be considered.

The more proxy services a proxy has, the better since this saves time in downloading from elsewhere. The smaller RTT will result in the lower communication delay between entities in the system.

Every proxy has a local repository of proxy services' archive files. A proxy is said to have a proxy service, if the archive file of that proxy service exists in the local repository of the proxy.

The proxy uses ping calls to find the RTT to the mobile device and the remote service. The IP addresses of the mobile device and the remote service should be included in the request sent by the *PFS* to the proxy.

It is assumed that the list of available proxy services is a static attribute, while the RTT between proxy and mobile device and between proxy and remote service is taken as a dynamic attribute. This means that the *PFS* knows the list of available proxy services at all of the registered proxies, but needs to inquire proxies about the values of RTT to mobile device and the remote service.

As mentioned in the discussion of algorithm 1, each proxy is shown as a list of numeric values for attribute. The RTT attribute inherently has numeric values, but there is a need to represent the other attribute, e.g., the list of available required proxy services numerically. This attribute is quantified by calculating the percentage of required proxy services that are available on the proxy.

4.4 Experiments

This section presents experiments used for evaluating the proxy-based architecture. The metrics used to evaluate performance are presented in section 4.4.2. A description of the remote services and the mobile application used in the experiments are provided in sections 4.4.4 and 4.4.5. The experimental set up is described in section 4.4.3. The description of the experiments, results and evaluation of the results are presented in sections 4.4.6-4.4.8.

4.4.1 Design Goals

The experiments are designed to provide insight into the following questions:

1. Are proxies, as defined in the proposed proxy-based system, useful in improving the quality of the service provided for mobile applications, in the mobile computing environment?

2. How useful is the proposed proxy discovery mechanism?

3. Is the overhead incurred by using proxies tolerable for mobile applications?

4. What is the impact of factors used in the proxy discovery process on the performance of mobile applications?

4.4.2 Evaluation Metrics

The metric used for measuring the performance in the experiments is the runtime of the mobile application. In the experiments, the runtime of an applications that uses the proxy services is compared to the runtime of an application that does not use any proxy services. Both applications call the same remote services. It is aimed to find the effect of using proxy services on the runtime, by comparing these applications.

The overhead added by using the proposed proxy-based system is measured as well. The overhead is defined as the additional execution time of the mobile application incurred when using proxy services.

4.4.3 Experimental Setup

The mobile application used for the experiments related to *Relay* proxy service, presented in sections 4.4.6.2 and 4.4.6.4 is implemented for an HTC Magic cell phone, with Android OS v2.1, 288 MB memory and 528 MHz CPU. The mobile application used for experiments related to *DataTransfer* proxy service, presented in sections 4.4.6.3 and 4.4.6.4, is implemented for an Acer Iconia Tab A500 with the Android 3.2.

The *Printing service*, *Printer Finder*, proxy services, and the *PFS* are implemented as web services with Apache Axis2 1.6, as the web service engine. Proxy services are hosted on a single host which is the proxy machine. Other services are hosted on distinct servers. The specification of all the server machines are identical with 2.4 GHz CPU and 2 GB RAM. The Apache Tomcat 7.0 application server is installed on all servers. A schema of the components used in the experiments is shown in diagram 4.2. As shown in the diagram all servers are on the same local area network.

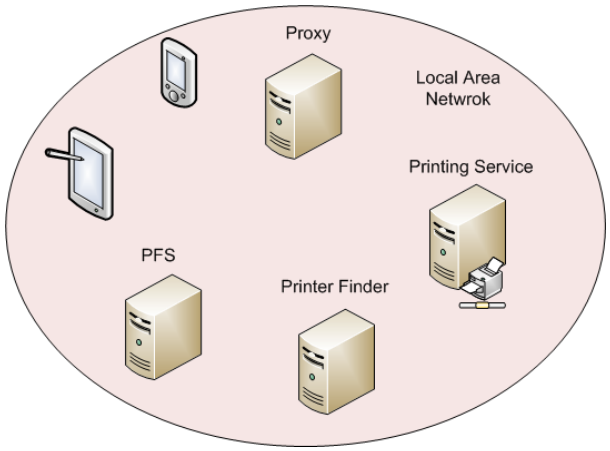


Figure 4.2: Experimental Setup for Proxy Discovery Experiments

The server machines are all connected to an Ethernet 100 Mbps LAN and the cell phone is connected to the LAN through IEEE 802.11g access points.

4.4.4 Remote Services

The application chosen for the experiments is remote printing, which was introduced in section 3.3.1. This application involves two remote services: *Printer Finder* and *Printing Service*.

The *Relay* proxy service, which provides disconnection handling, between mobile application and remote services with longer response times, can be used by mobile applications to call the *Printer Finder*, as shown in figure 3.3.

The *DataTransfer* and *CodeExec* proxy services can be used for offloading the task of calling the *Printing Service* to the proxy, as shown in figure 3.4. As described in section 3.3.1.2, the *DataTransfer* proxy service provides support for disconnection handling during the transmission of the client code of the *Printing Service* and the file to be printed from mobile device to the proxy, and the *CodeExec* proxy service can be used for the execution of the *Printing Service* client on the proxy.

The implementation of the *Printing Service* is a stub for the remote printing task. Only the file transmission functionality of this service is implemented. This is acceptable since only the time to transfer files is measured in the experiments.

4.4.5 Mobile Applications

Two mobile applications were used in the experiments. The first mobile application, *RemotePrintViaProxy*, uses the proposed proxy-based system. This mobile application calls the *Printer Finder* service through the *Relay* proxy service, and calls the *Printing Service* with the use of the *DataTransfer* and *CodeExec* proxy services. The diagrams show the interaction between this mobile application, the proxy services, and the remote services are presented in section 3.3.1.3 (figures 3.3 and 3.4).

The second mobile application, *RemotePrintWithoutProxy*, accesses the remote services directly. This mobile application does not use the proposed proxy-based system. The diagram showing the interaction between the *RemotePrintWithoutProxy* application and remote services is shown in figure 4.3.

The comparison of these two mobile applications is used to determine the effectiveness of the proposed proxy-based system with respect to performance.

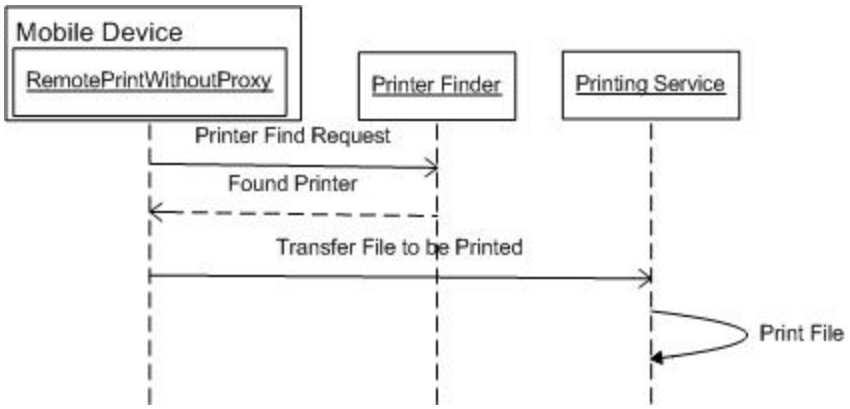


Figure 4.3: Call *Printer Finder* and *Printing Service* Without Proxy

4.4.6 Using Proxy or Not Using Proxy: Retransmission vs Recovery

As noted earlier, mobile applications often make use of a remote service that is often hosted on a cloud. This works well when the application is not highly interactive and relatively little data is transferred. The effectiveness of the proposed architecture is intended to support applications that have different requirements than the current suite of mobile applications.

The evaluation of the effectiveness of the architecture is based on a mobile application that requires relatively large amounts of data to be transferred to a remote service and is sensitive to disconnections especially when the connection is made to a remote service that may need a considerable amount of time to process a request from the mobile application.

As explained in chapter 3, the *Relay* proxy service is designed to handle disconnections between the mobile application and the remote service and the *DataTransfer* proxy service is designed support the handling of disconnections when large amounts of data is being transferred.

The evaluation is done by comparing the performance of the *RemotePrintViaProxy* and the *RemotePrintWithoutProxy*.

The first application, *RemotePrintViaProxy*, introduced initially in section 3.3.1.3, uses the *Relay* proxy service to call the *Printer Finder* remote service, and uses the *DataTransfer* to call the *Printing Service*. The second application, *RemotePrintWithoutProxy*, which is initially introduced in section 4.4.5, directly calls both remote services, without help of proxy services.

The effect of the two proxy services, on the performance of mobile applications, are examined in separate experiments. In the first set of experiments, section 4.4.6.2, the *RemotePrintViaProxy* and the *RemotePrintWithoutProxy* that call the *Printer Finder* service are compared. This comparison is used to determine the effectiveness of proxies that host a service that deals with disconnections for applications that are sensitive to disconnections. In the second set of experiments, section 4.4.6.3, the *RemotePrintViaProxy* and the *RemotePrintWithoutProxy* that

call the *Printing Service* are compared. This comparison is used to determine the effectiveness of proxies that host a service that deals with disconnections for applications that transfer large amounts of data.

4.4.6.1 Disconnection Simulation

In the experiments, disconnections occur during file transmission. File transmission is implemented using socket communication. The server socket is running on the proxy machine and the client socket is running on the mobile device. A disconnection is simulated by deliberately closing the socket at the client side, after sending half of the bytes of a file. The server socket at the proxy machine, which is listening on the connection, discovers this connection termination using a timeout. Afterwards, the server socket is terminated.

4.4.6.2 Calling *Printer Finder*

The *RemotePrintViaProxy* application and the *RemotePrintWithoutProxy* application are executed three times. For the *RemotePrintViaProxy* application, a disconnection is simulated after calling the *Relay* proxy service and before receiving the result. For the *RemotePrintWithoutProxy* application, disconnections are simulated after calling the *Printer Finder*, and before receiving the result.

After discovering the disconnection and reconnecting, the *RemotePrintViaProxy* contacts the *Relay* proxy service to retrieve the results. However, the *RemotePrintWithoutProxy* application has to re-call the *Printing Finder*, since the *Printer Finder* has no mechanism to save the result for its client if a disconnection occurs.

For the *RemotePrintViaProxy*, after discovering the disconnection, the time to contact the *Relay* proxy service, retrieve the result, and then notify the *Relay* proxy service of the successful receipt of the result, is recorded for the three runs. The average time over three runs is 759.33 msec.

For the *RemotePrintWithoutProxy* application, after discovering a disconnection, the average time to repeat the call to the *Printer Finder*, and to receive the result from the service is recorded in all the three runs and the average is 8.70 seconds.

As the numbers show, retrieving the result from the *Relay* proxy service is much faster than the re-calling of the remote service in this case.

4.4.6.3 Calling *Printing Service*

The *RemotePrintViaProxy* application and the *RemotePrintWithoutProxy* application are executed three times, while calling the *Printing Service*. A disconnection is simulated during the

transmission of the file for all runs. The disconnection is simulated after transferring half of the bytes of each file for all runs.

For the *RemotePrintWithoutProxy*, the file transmission occurs between the mobile application and the *Printing Service*, while for the *RemotePrintViaProxy* the file transmission is between the mobile application and the *DataTransfer* proxy service. Upon reconnection, the *RemotePrintViaProxy* mobile application retrieves the number of successfully transmitted bytes from the *DataTransfer proxy service* and resumes the transfer of the remaining bytes of the file. On the other hand, the *RemotePrintWithoutProxy* mobile application has to re-send the entire file, from the beginning, after the reconnection.

The comparison of the performance of *RemotePrintWithoutProxy* and *RemotePrintViaProxy* after a disconnection is done by comparing the the recovery time of the *RemotePrintViaProxy* and the time it takes to re-transfer the already transferred bytes for the *RemotePrintWithoutProxy*. For the *RemotePrintViaProxy*, the recovery time is defined as the time between the reconnection and the time that the application has found the number of already transmitted bytes from the *DataTransfer* proxy service, and is available to start the transmission of the the remaining bytes of the file.

Experiments are run for various file sizes. This provides insight into how file size impacts the use of a proxy. The results of the experiment are shown in table 4.1. The calculated times are averaged over three runs for each file size. The standard deviations are presented in Appendix A, table A.1. As shown in the table, the recovery times achieved by the *RemotePrintViaProxy* are much shorter than the re-transmission times of the *RemotePrintWithoutProxy*, even for the smaller data sizes. This suggests that using the *DataTransfer* proxy service for mobile applications that need to transfer data with various sizes to remote services is highly beneficial when a disconnection occurs.

We also investigated placing the functionality associated with the *DataTransfer* proxy service as part of the implementation of the *RemotePrintWithoutProxy* and have the *RemotePrintWithoutProxy* directly communicate with the *Printing Service*. However, for recovery purposes the *Printing Service* should be able to provide the *RemotePrintWithoutProxy* with the information required for disconnection handling, i.e. the number of bytes successfully received by the *RemotePrintWithoutProxy*. This means that the *Printing Service* has to be modified to support keeping track of this information for each print job. When using the functionality provided by the *DataTransfer* proxy service, there is no need to modify the *Printing Service* to support mobile devices.

Table 4.1: Retransmission (RemotePrintWithoutProxy) vs. Recovery (RemotePrinteViaProxy)

File Size	Re-transmission Time for *RemotePrint-WithoutProxy* (msec)	Recovery Time for *RemotePrint-ViaProxy* (msec)
300 KB	120.50	19.33
600 KB	198.17	29.33
900 KB	217.00	46.33
3 MB	593.17	110.67
5 MB	931.67	176.00
10 MB	1807.83	343.00
15 MB	2562.00	506.33
20 MB	3115.50	601.33
25 MB	3829.33	847.67

4.4.6.4 Measuring the Overhead

The overhead of using the proposed proxy-based system is evaluated by comparing the total execution time of the *RemotePrintViaProxy* and the *RemotePrintWithoutProxy* application when they call remote services and no disconnection happens during the calls. A longer execution time for mobile applications that use proxy-services is expected since the call to remote services passes through proxy services, while mobile applications that do not use proxy-based system, directly call the remote service.

To measure the overhead time added by using the *Relay* proxy service, both mobile applications call the *Printer Finder* service. No disconnections are simulated. The run-time of the *RemotePrintViaProxy* application, averaged over three runs, is 9.03 seconds. The average run-time of the *RemotePrintWithoutProxy* application over three runs, is 8.70 seconds. The small difference between the two times suggests that using the *Relay* proxy service does not add considerable overhead to the run-time of applications.

To measure the overhead added by using the *DataTransfer* proxy service, both applications called the *Printing Service* and no disconnection was simulated during the execution of applications. The *RemotePrintViaProxy* and *RemotePrintWithoutProxy* tried to print several files with various sizes. The execution times for both mobile applications are recorded. The results are shown in table 4.2, with standard deviations presented in table A.2 of appendix A. The results show that the *RemotePrintViaProxy* took longer to finish file transmissions compared to *RemotePrintViaProxy*. However, as the numbers on the fourth column, *Overhead* show, the difference between the execution time of the two mobile applications do not increase as the file

size increases. This result indicates the suitability of using the *DataTransfer* proxy services for larger file sizes, especially by taking into consideration the effectiveness of *DataTransfer* proxy in handling disconnections, as shown in table 4.1.

Table 4.2: Execution Time with No Disconnection and the Overhead

File Size	Execution time of *RemotePrint-WithoutProxy* (sec)	Execution time of *RemotePrinte-ViaProxy* (sec)	Overhead (sec)
300 KB	0.24	3.50	3.26
600 KB	0.40	3.55	3.15
900 KB	0.43	3.67	3.24
3 MB	1.19	4.50	3.31
5 MB	1.86	5.06	3.20
10 MB	3.62	7.12	3.50
15 MB	5.12	9.00	3.88
20 MB	6.23	10.24	4.01
25 MB	7.66	11.22	3.56

4.4.6.5 Adaptive Approach

The results suggest that proxies should not be used for transferring small files. The client applications can be designed in a way to be able to choose between using proxy services, or access the remote service directly, depending on the amount of data to be transferred.

A mobile application, named *AdaptiveRemotePrint*, is designed to decide on using or not to use the proxy based on the size of the file to be transferred. The *AdaptiveRemotePrint* application uses the *DataTransfer* proxy service only when that the size of the file is larger than five MB. Otherwise, *AdaptiveRemotePrint* application directly calls the *Remote Printing* service.

The *AdaptiveRemotePrint* application is executed three times for each file size and the result obtained from averaging over three runs is shown in table 4.3. Standard deviations are presented in table A.3 in appendix A. As expected, the achieved result for files equal or smaller than five MB is similar to result of *RemotePrintWithoutProxy*, while the execution time for files larger than five MB is similar to the result of *RemotePrinteViaProxy* (table 4.2). The higher execution time for files larger than five MB is acceptable by taking into consideration the higher probability of disconnection occurrences, and the ability of the *DataTransfer* proxy service in fast recovery from disconnections as shown in table 4.1.

In the end, the use of proxy service depends on the application's tolerance for frequent

Table 4.3: Execution Time of AdaptiveRemotePrint Application

File Size	Execution time of *AdaptiveRemotePrint* (sec)
300 KB	0.24
600 KB	0.40
900 KB	0.43
3 MB	1.15
5 MB	1.74
10 MB	6.74
15 MB	8.12
20 MB	10.12
25 MB	11.27

disconnections e.g., for this example application, disconnections during small files transfers are tolerable, but not during large files transfers. As a result, the mobile application might use the *DataTransfer* proxy service during the transmission of large files.

4.4.7 Proxy Discovery: Effect of Preferences

As mentioned in section 4.2.3, the components in the system, i.e. the *PFS*, proxy, proxy services are designed to consider multiple attributes for choosing the best proxy. This section shows the impact that user preferences have on proxy selection. The implemented prototype of the proposed system uses three attributes: i) the percentage of required proxy services available on the proxy, ii) the proximity of the mobile device and the proxy, and the proximity of the proxy and the remote service, if there is a need to access a remote service, and iii) the CPU load of proxies.

4.4.7.1 Proximity

The proximity between components over a network can be evaluated by the geographical distance between components or can be measured using the RTT between components, or the hop count between them. In our prototype implementation we use RTT values. Smaller RTT values between two components means less communication delay between those components. The RTT between two nodes changes over time as a result of the change in the load over the route between nodes. As the traffic between nodes increases, the length of router queues over the route increases and this often will lead to higher RTT. As a result, the RTT can be viewed as a

snapshot of the state of the route between nodes. Our use of RTT is motivated by other work that uses RTT, e.g. [56, 59], as a distance measure.

4.4.7.2 Effect of RTT between Mobile Device and Proxy

The experiments in this section has the *RemotePrintViaProxy* mobile application communicate with the *Remote Printing* service. It is assumed that all required proxy services are available on all proxies and the CPU load on the proxies is not taken into consideration. This means that the only factor used in selecting a proxy is the proximity. This is an acceptable condition since the *Remote Printing* application occasionally involves the transmission of large files from the mobile device to the remote service and as a result the response time of the application depends on the RTT between mobile device and proxy.

RTT Simulation and Assumptions

The experimental environment is described in section 4.4.3. The challenge with experiments results from the use of the Western campus network for communication between the cellphone and the servers. We do not have control over network load and hence there is a need for RTT simulation.

It is assumed that the RTT affects only the file transmissions from mobile applications. This means that we assume that the RTT does not affect the call to services when:

1. no file transmission is involved in calling the service, and
2. calling the service involves file transmission, but the client and the service are running on wired machines

The assumptions are for experimental purposes. We are primarily interested in the RTT incurred by wireless communications.

Several experiments were carried out to find the approximate RTT between components of the system, i.e. mobile device, machines hosting proxies, and the server hosting *Printing Service*. In these experiments the result of the ping command between machines hosting proxies and *Printing Service* showed that RTT values are negligible, i.e. less that 4 milliseconds. The reason is that all of these machines are connected to the same LAN. Consequently, it was decided to ignore the real RTT between wired components when gathering results during the experiments.

During the execution of *RemotePrintViaProxy* file transmissions occur between the *RemotePrintViaProxy* mobile application and the *DataTransfer* proxy service on the proxy, and between the *DataTransfer* proxy service and the *Printing Service*.

Although calling the service involves file transmission between the proxy and the server running the remote service, we do not consider RTT since the proxy and the server are on the same wired network. Furthermore since the proxy machine and the server with the remote service are on the same LAN there is little variability in the time it takes to transfer a file. RTT simulation is done for file transmission from the mobile application to the *DataTransfer* proxy service when executing the *RemotePrintViaProxy*. In addition, the RTT is not simulated in the call of the *CodeExec* proxy service from the *RemotePrintViaProxy* since no file transmission is involved. Likewise, no call to the *Printing Finder* is done in the experiments presented in this section since no file transmission is included in calling the *Printer Finder* service,.

Socket programming is used for file transmission in the implementation of the *DataTransfer* proxy service. The server socket is part of the *DataTransfer* proxy service on the proxy, while the client socket is part of the API provided for the mobile application and resides on the mobile device. To simulate the RTT, the client socket, on the mobile device, sleeps for the simulated value for the RTT, before every write to the connection.

Experiments

The experiments presented in this section shows that impact of RTT on application performance.

The values used for the simulation of RTT are shown in the first column on table 4.4. These values are chosen based on several experiments that were performed in order to estimate the value of RTT from the mobile device to remote machines located at various locations. As expected, it was shown that RTT values between the mobile device and a remote host is not fixed over the time and fluctuates in a range. As a result, it is more realistic to generate RTT values from a range, instead of using a fixed value. Consequently, the RTT values are randomly generated from a range to simulate the fluctuations in the RTT in the real world. Based on the initial experimental results, it is assumed that in a steady state environment that the load of data transmission in communication links does not drastically change. Thus we have RTT values changing in intervals of length 5 msec. The intervals are shown in the first column of the table 4.4, and the average of the RTT values, generated from each interval, are calculated and shown in the second column of the table.

The *RemotePrintViaProxy* application is executed five times. The time to call the *Printing Service* is measured for each run. The result, that is averaged over five runs, is shown in the third column of table 4.4. As can be seen from table, higher RTT values resulted in higher execution times. This shows that it is beneficial to use proxies that are closer to the mobile device, i.e. have smaller RTT to the mobile device. The results show that for this application,

which involves data transmission between the mobile application and the proxy, it is beneficial to have a proxy discovery mechanism that finds a proxy with the smallest RTT.

Table 4.4: Time for Calling Printing Service with Various RTT values

Simulated RTT Range (msec)	Average Simulated RTT (msec)	Time for Calling Printing Service (sec)
[40, 45]	42.18	172.01
[35, 40]	36.89	151.46
[30, 35]	32.40	135.12
[25, 30]	26.77	113.54
[20, 25]	21.88	95.23
[15, 20]	16.51	73.70
[10, 15]	11.71	53.30
[5, 10]	7.42	35.80
[0, 5]	0.84	10.21

4.4.7.3 Effect of Proxy Discovery Factors

Mobile application requirements should influence the proxy discovery operation. For example, as shown in the experiments in section 4.4.7.2, the RTT factor is important when choosing a proxy for *RemotePrintViaProxy* application.

The experiments described in this section shows the importance of selecting correct factors for the proxy discovery operation. The mobile application used for the experiments, *MobileMatrixMultiplicationClient*, is designed in a way to demand high processing power. This application calls a remote service named *Matrix Multiplication* service that simply performs the multiplication of two matrices with 1500 rows and columns. It is assumed the matrices are created by the service and no data transmission from the mobile application to the service is required.

This application is an example of applications that require a high amount of computational resources but do not involve large amounts of data to be transmitted. Applications with these characteristics include search applications and translators.

This kind of application requires a service on a machine with high processing power. To find the importance of offloading this service from mobile devices a mobile application with the same functionality, i.e. multiplication of two matrices with 1500 rows and columns, is implemented and executed three times on the Acer tablet. On average it took more than eight minutes to finish the matrix multiplication on the Acer tablet. This long runtime on a mobile device suggests offloading this service to a proxy is a good option for this application.

For this application, the *CodeExec* proxy service can be used to deploy and access the *Matrix Multiplication* service on the proxies. However, during the experiments it is assumed that the service is already installed on all proxies and the mobile application directly calls the service. As a result no proxy services is required for the functionality of *MobileMatrixMultiplicationClient* application.

Two proxy machines, *Proxy A* and *Proxy B*, are used in the process of proxy discovery and various CPU loads are generated to study the impact of proxy CPU load on the performance of application.

CPU Load Simulation

To generate CPU load on a proxy machine the lookbusy [6] tool is used. Using this tool a specified CPU load percentage, for the desired time period, can be generated on a machine.

Experiments

For all the experiment runs in this section a constant load of 10% is genarated on the *Proxy B*, while various loads, from 10% to 90% are generated on the *Proxy A*. Since the goal is to study the impact of the proxy CPU load, for all the experiments it is assumed that the value of RTT between mobile device and the two proxies are equal.

In the first set of experiments, the weights assigned to the proxy discovery factors are as follows:

- RTT between mobile device and proxy: 0.5
- Proxy CPU load: 0.0
- Available proxy services: 0.5

With this weight assignment, and with the assumption that no proxy services are required, and the RTT between mobile device and both proxies are equal, the calculated appropriateness of both proxies will be equal, even if the CPU load of proxies are different since the weight of zero is assigned to the proxy CPU load. With this setting it is expected that regardless of the CPU load of proxies, the first proxy in the list of proxies at the *PFS* be always chosen.

With these weights the mobile application is executed three times. The mobile application first contacts the *PFS* to find a proxy. After finding the proxy, it calls the *Matrix Multiplication* service on the proxy. The time between the start of the proxy discovery operation and completion of service is recorded. The CPU load generated on the proxies, the average runtime of

three runs, and the chosen proxy are shown in table 4.5. The obtained standard deviation are shown in appendix A, table A.4. As expected, for all three runs and for all CPU load generated, always *Proxy A* is chosen by the *PFS*. In addition, the runtime increases proportionally with the CPU increase on the *Proxy A*.

Table 4.5: Time for Calling Matrix Multiplication Service - Proxy CPU Load is not a Proxy Discovery Factor

Proxy A CPU Load (%)	*Proxy B* CPU Load (%)	MobileMatrixMultiplication Runtime (sec)	Chosen Proxy
10	10	48.76	*Proxy A*
30	10	48.59	*Proxy A*
50	10	49.24	*Proxy A*
70	10	54.34	*Proxy A*
90	10	63.16	*Proxy A*

In the second set of experiments a different weight assignment is used as follows:

- RTT between mobile device and proxy: 0.3
- Proxy CPU load: 0.3
- Available proxy services: 0.4

In this weight assignment, the weight assigned to available proxy services does not have any effect since no proxy services are required. Equal weights are assigned to the RTT and proxy CPU load, but since it is assumed that RTTs between the mobile device and both proxies are the same, the RTT does not have any effect on the proxy discovery. As a result, it is only the value of proxy CPU load that is effective in choosing the proxy. The same CPU loads as used in the previous set of experiments are generated on proxies. The runtime of mobile application and the chosen proxy are shown in table 4.6.

Table 4.6: Time for Calling Matrix Multiplication Service - Proxy CPU Load is a Proxy Discovery Factor

Proxy A CPU Load (%)	*Proxy B* CPU Load (%)	MobileMatrixMultiplication Runtime (sec)	Chosen Proxy
10	10	47.94	*Proxy A*
30	10	48.474	*Proxy B*
50	10	48.349	*Proxy B*
70	10	49.077	*Proxy B*
90	10	51.169	*Proxy B*

As expected, the proxy discovery mechanism, always chooses the proxy with lower CPU load (For the experiment that the load on *Proxy A* and *Proxy B* is 10%, i.e. the first row of table, the spontaneous measured load on *Proxy A* is slightly smaller than the spontaneous meseared load on *Proxy B*, as a result *Proxy A* is chosen.) As can be seen from tables 4.5 and 4.6, the runtime of the mobile application for all runs in table 4.6 is close to the runtime of the mobile application in table 4.5, when *Proxy A* is chosen and the CPU load of *Proxy A* is 10%, i.e. the first row. The reason is that in all of these cases the service is called and executed on a proxy with 10% CPU load. The results presented in this section show the impact of weights assigned to factors where the weights and factors are assume to represent preferences.

4.4.7.4 Overhead of Proxy Discovery Operation

The proxy discovery process adds overhead to the overall response time of the system. To measure this overhead the time to find a proxy in almost all experiments is recorded. The results shows that the attempt to find a proxy takes about 0.6 seconds.

As the number shows, the overhead is not significant, compared to the entire time of calling remote services, for proxies with higher RTTs, as shown in the third column of table 4.4 and for proxies with higher CPU loads as shown in the third column of table 4.5.

4.4.8 Conclusion

The experiments presented in this section were designed to answer two questions: i) Is the proposed proxy discovery architecture effective, and ii) Does the proxy-based system improve performance and what is the overhead?

The results of experiments show that the mobile applications, that are sensitive to the RTT to the proxy, perceive better quality when using proxies with smaller RTT. As a result, the proposed proxy discovery mechanism that is able to find a proxy with smallest RTT can be effective. In addition, the proxy-discovery system works satisfactory for applications that need to have access to proxies with lower CPU load. It generally shows that the proxy discovery mechanism that is able to take into consideration the preferences of the mobile applications, can effectively improve the performance of the mobile application.

The result of the experiments also show that the proxy services implemented as part of the prototype of the proposed infrastructure were successful in providing the disconnection handling facilities when mobile applications access remote services. The results obtained from experiments show that using the mobile computing infrastructure adds a run-time overhead when calling to remote service involves data transmission and no disconnection happens during the call the remote service. Although, the communications in the wireless environment

are prone to frequent disconnections, the use of the proposed proxy-based infrastructure is beneficial, even in the presence of the mentioned overhead.

The percentage of the consumed battery energy was measured for the experiments and it was found negligible, i.e. less than one percent. This shows that the energy consumption of operations of the proxy-based system and the proxy discovery mechanism is minimal.

4.5 Summary

The proxy discovery method, designed for the proposed proxy-based mobile computing environment 3, is presented in this chapter. The discussion on the related work, presented in section 4.1, shows several shortcomings of other proxy-based approaches in finding an appropriate proxy for a mobile application, such as scalability, not being aware of the changing condition of the mobile computing environment, not taking into consideration the preferences of the mobile applications, etc. In this chapter it was shown how the proposed proxy-discovery can answer the problems existing in other proxy-based systems. The detailed description of the interactions happen between components of the system, i.e. mobile application, the *PFS*, and proxies is provided. Additionally, the algorithms used by the *PFS* to choose the most appropriate proxy for mobile application are presented. To find the effectiveness of the proposed proxy discovery mechanism, several experiments are designed and executed. The results of the experiments show that the proposed approach positively impacts the performance of the mobile applications, when transferring data over bandwidth limited wireless links, and especially in the presence of disconnections, that are perfectly frequent in wireless environment.

Chapter 5

Handoff

Chapter 4 describes algorithms used to associate a mobile device with a proxy. The proxy selection decision is partially based on the use of dynamic attributes that characterize proxy behaviour e.g., load. It is possible that resource availability on a proxy changes over time e.g., an increase in CPU usage may negatively impact the performance of the proxy services running on the proxy. Another change is that the wireless connection to the proxy may change as a mobile device moves resulting in weakened signal strength.

It may be desirable for a mobile device to associate itself with a different proxy due to changes in the connectivity or the resource availability in the proxy. We propose the use of a handoff operation that allows for a dynamic change in the proxy associated with a mobile device.

In this chapter the work in proxy-based mobile computing systems that include handoff and migration of clients are presented. The details of the handoff operation in our proposed system is presented in the section 5.2 is described. Several experiments are designed and run to examine the effectiveness of handoff process. The example AR application, introduced in section 3.3.2, and a prototype of the proposed proxy-based system are implemented to be used for the experiments. The details of the experiments and the achieved results of experiments are presented in section 5.3.

5.1 Related Work

The users of mobile devices are free to move, the condition of the network may change, and the load on the proxy varies over time. These changes may result in a situation where the proxy that was suitable at the start of the communication is no longer available. Little of the work that proposes the use of a proxy considers changes in the environment. Examples of this work

are presented in this section.

5.1.1 Mobile Service Clouds

An infrastructure for autonomic communication service is introduced by Samimi et al. [59]. The goal of the autonomic communication services is to dynamically reconfigure and instantiate services based on network conditions. The reconfigurations and instantiations are transparent to the end applications, i.e. the clients of services. Examples of these services are services that support fault tolerance, enhance security, and improve the quality of service provided for clients. The proposed infrastructure, named *Service Cloud*, is composed of a middleware layer and an overlay network.

The system was initially introduced in an earlier paper, McKinley et al. [50]. The work in Samimi et al. [59] focuses on expanding the system for supporting applications on mobile devices. It is assumed that mobile devices are located at the edge of the network and are connected to the nodes on the overlay network that are within few hops. The service provided for mobile applications should be aware of the characteristics of the mobile device. This includes the battery lifetime, processor speed, memory size, display characteristics, etc. The example service used in the experiments is a data streaming service. The services developed to enhance the quality of service received by the mobile application can be installed on the nodes of the service cloud. Examples of services that are specifically useful for mobile application are data transcoding, handling frequent disconnections, and enhancing the quality of the connection using Forward Error Correction (FEC) technique.

The *Service Cloud* is composed of two parts: i) the *mobile service cloud*, and ii) the *deep service cloud*. The *deep service cloud* is made of a collection of nodes that host the middleware and are connected together by the overlay network. The *mobile service cloud* is added to the *deep service cloud* to support mobile applications and includes hosts that are connected to one network, for example the network in a university campus. The services that should provide real-time response can be installed on the mobile service cloud. As a result, those services are close to the clients connected to the same network.

Another advantage of hosting services close to clients is that it reduces the network traffic. For example, a service that implements forward error correction (FEC) method can be installed closer to its mobile clients. As a result, the extra data generated by the FEC method is transferred between the FEC service and the mobile client, rather than between the source of data and the client.

One of the nodes on the overlay network, which is called the *Service Gateway*, is responsible for finding a primary proxy for clients. The client needs to know the address of the *Service Gateway*. Upon receiving a request from a mobile application, the *Service Gateway* selects a node on the *deep service cloud* as the primary proxy. Afterwards, the primary proxy chooses the transient proxy on the *mobile service cloud*. The transient proxy is called transient since it can migrate along with the mobile device to other networks. The route between the server, primary proxy, transient proxy, and mobile application is established and required services are installed on proxies.

In the experiments, the service is a UDP streamer. The primary proxy monitors the activities of the transient proxy. If the transient proxy fails, the primary proxy dynamically chooses a new transient proxy on the *mobile service cloud* and redirects the data stream to it. In this paper mobility is not investigated in the discussion and experiments. The experiments are designed for the situation that the transient proxy fails.

In another work Samimi et al. [58] introduce three case studies and have shown their service cloud infrastructure can be used in those case studies. One of the case studies is about providing video streaming service for a mobile node in the wireless part of network. The transient proxy on the *mobile service cloud* provides two services, multicasting and FEC. The requested video stream is initially unicasted from the server to the transient proxy. The transient proxy is able to multicast it to multiple clients on the wireless network in its vicinity.

In this use case the functionality of the infrastructure in handling the mobility of a mobile device is investigated. In this work the movement of the mobile device is discovered by the change of IP address of the mobile device. A change of IP address occurs when the mobile device leaves a network and enters another network. In the case study, a laptop is initially connected to the wired LAN. The laptop then disconnects from the LAN and start to use a Wi-Fi connection. This change of network causes the change of IP address. It is the responsibility of the mobile device to inform the primary proxy about the change of its IP. The primary proxy finds a new transient proxy to service the mobile device and redirects video stream to it.

5.1.2 Change of Proxy in Data Link Layer Handoff Manager

A self-adaptive data link layer handoff management, using a proxy, proposed by Bellavista et al. [16], is presented in section 2.1.2.2. The change of proxy in this work, as the result of mobility, is discussed by Bellavista et al. [17]. It is assumed that all wireless LANs under the same administration authority are grouped in a domain and each domain is managed by a *service gateway*. For each mobile device in the domain, a *service proxy* is created on the *service*

gateway. The *service proxy* is responsible for the management of data link layer handoff of its associated mobile device.

The system supports the mobility and movement of the mobile device. This means that if the mobile device moves to a another domain, the *service proxy* of the mobile device is migrated to the *service gateway* of the new domain. The adaptation to the movement of mobile devices and migration of *service proxies* are supported by the *Fast Service Rebinder* method introduced in the paper.

The handoff at the data link layer is categorized into three categories: micro, macro and global handoff. During a micro handoff, the mobile device is moved between access points in the same network. As a result, the IP address of the mobile device does not change. During a macro handoff the IP address of the mobile device changes as a result of the mobile device leaving a network and entering a different network. During a global handoff, the mobile device has moved to a network managed by another administration authority. In this case, not only does the IP address of a mobile device change, but also the the Authentication, Authorization and Accounting (AAA) data of the mobile device and the user should be transferred to the administration authority of the destination domain.

The *Fast Service Rebinder* method is able to handle macro and global handoff. During a macro or global handoff, the *service proxy* is migrated to the new domain. In the case of the global handoff, the context information related to the proxy, including the AAA data should be transferred as well. The *service proxies* can be used for data buffering. When using the proxy for buffering purposes, the buffer of the *service proxy* should also be transferred to the target domain.

For proxy migration and context transfer, a SIP basic mechanism is employed. The migration of a *service proxy* involves the activation of the *service gateway* at the target domain. The *service proxy* on the new domain, after being activated, contacts the proxy on the old domain and asks for the transfer of the context data and the data in the buffer.

5.1.3 Task Transfer between Cloud Providers

In the "Economic Mobile Cloud Computing" system (Liang et al. [48]), presented in section 2.1.1.3, the weblets, that are computation units, offloaded from mobile devices, might be transferred between service nodes (SNs) in the same cloud domain or in another domain. The transfer of weblets is based on the following: i) maximal could domain revenue and cost, ii) service expenses of mobile device, and iii) energy conservation mobile device. There is no discussion on the state transfer during weblet transfers.

5.1.4 Mobility Support in Proximal Community Network

In the work of Fesehaye et al. [25], presented in section 2.2.3, a mechanism to support the movement of mobile devices, during the use of a service on a cloudlet, is provided. When the mobile device moves to a location far from the servicing cloudlet, and closer to another cloudlet, the mobile device connects to the closer cloudlet. The newly connected cloudlet, is able to relay messages between the mobile device and the servicing cloudlet. As a result the use of the service by the mobile device can continue. The advantage of this approach is that no task migration or state transfer is required. Although, as the results of experiments show, if mobile device moves too far and the number of cloudlets between the mobile device and the servicing cloudlet increase, the quality of service degrades.

5.1.5 Gap Analysis

As mentioned previously, the proxy-based systems usually do not provide mechanisms to adapt to changes in the environment. Only a few provide techniques to react to changes in some factors. The mobility of the mobile device in Bellavista et al. [16] and Samimi et al. [59] which is defined as the change of IP address and is resulted from leaving a network and joining another network, is supported by moving the proxy along with the mobile device to the new network. Deploying the proxy in the network that the mobile device is connected to reduces the communication latency and data traffic. Although, in more general solutions, it might not be possible as the result of not having administrative access to the network that the mobile device is connected to and as a result not being able to deploy the proxy on the newly connected network.

In Fesehaye et al. [25] the cloudlets are equipped with a specific hardware that acts similar to access points. Mobile devices need to be present in the area covered by the cloudlet to be able to use it. This condition limits the mobility of the mobile device and limits the usability of the cloudlets.

In Liang et al. [48] no discussion on the mobility is provided. The task migration is done to seek more processing power. Weblets are transferred from the mobile device to a cloud, or transferred between clouds when more processing power is needed.

The work described above provides some support for adapting to changes in the environment. However, this support is limited in some aspects and are applicable to some specific systems. For example, the proposed method in Fesehaye et al. [25] mandates that proxies have the capabilities of wireless access points, or the work in Samimi et al. [59] and Bellavista et al. [16] requires full control on all networks that mobile device would connect to. Methods are needed that do not add limitations, while providing adaptation mechanisms when the

environment changes.

There is also a need to include the capability of adapting to the condition of the proxies, as provided by Liang et al. [48]. The processing load on a proxy may increase and there may be a need to switch to another proxy with less load.

Additionally, the tasks and services that are migrated between proxies might have states. When migrating a task between hosts, it might be needed to transfer the state of the task as well. The task transfer [48] does not provide any state transmission.

In this chapter the handoff mechanism, added to our proxy-based mobile computing system, is introduced. This mechanism provides adaptation in response to changes in the network and proxies, in addition to having mechanisms for state transmission along with the task migration between proxies.

5.2 Handoff Operation

The handoff operation is used to change the proxy associated with a mobile device. This section describes the conditions that may start a handoff process, the details of the activities carried out during a handoff process, and the description of the state transfer between proxies. The common functionalities that proxy services should have to conform to the proposed handoff mechanism are provided as well. Finally, it is presented how the mobile application is notified on the outcome of the handoff process e.g., the address of new proxy, with least interruption.

5.2.1 Handoff Decision

The handover decision is used to determine whether and how to perform the handover. The handoff operation is triggered by an event. There are multiple events that can trigger the handoff operation. An event is a condition on one or more attributes that are monitored to determine whether or not a handoff is needed. The events can be grouped as follows:

- Mobile application-related: defined based on the requirements of the mobile application
- Proxy-related: defined based on the conditions of the proxy

The mobile application-related events are triggered based on the information provided by the mobile application at the time it is associated with a proxy. This information includes threshold values on specific attributes. If the value of an attribute is outside the range of acceptable values, then a handoff operation may be initiated.

The proxy-related events are triggered based on the resource availability on the proxy. For example, the proxy might want to keep its CPU utilization below 80%. As a result, the proxy

defines the threshold value of 80% for its CPU utilization, and a handoff operation is triggered if the CPU utilization on the proxy exceeds 80%.

Thresholds

An event consists of conditions on attributes i.e., a comparison of attributes to threshold values. These can be defined either by the client application or the proxy. For example, the client can declare a threshold value for the RTT, or a proxy may define a threshold for CPU load. If the RTT value or the CPU load exceeds the defined threshold, then the handoff operation may be invoked.

If the mobile application require conditions on some attributes, the mobile application can inform the proxy of these conditions by including the attributes and their threshold values in the *Hello message* sent to the proxy, in the Step 7 of the proxy discovery mechanism, as presented in section 4.2.2.

Monitoring Attribute Values

The handoff decision mechanism can be carried out by the mobile application or by the proxy. The handoff decision requires monitoring of attributes that can be used to characterize some aspect of network behaviour or proxy behaviour. Monitoring incurs the use of additional energy usage. We chose to place the handoff decision mechanism with the proxy. Placing the handoff decision mechanism with the mobile application increases the battery consumption.

5.2.2 Selecting the Mobile Application for Handoff

There are multiple strategies that can be employed for choosing a mobile application for handoff. In the simplest approach, the proxy may randomly choose a mobile application for handoff. Another approach may depend on the attribute that has a threshold value violated. For example, if the proxy CPU load exceeds the threshold then the proxy might choose a mobile application for handoff that uses the CodeExec proxy service. This assumes that executing a task on the proxy is done because it is too compute intensive for the mobile device. We assume that each proxy owner defines their handoff policies.

5.2.3 Handoff Management Process

The diagram showing the interaction between proxies, *PFS* and the mobile application during a handoff operation in depicted in figure 5.1.

Step 1

When the handoff operation is invoked, the proxy contacts the *PFS* to find a new proxy. The process of finding a new proxy during the handoff is the same as the process presented in the section 4.2 with the exception that the request is sent by the current proxy to the *PFS*, rather than the client application. The proxy retrieves the initial request of the client application from its local database, that includes the list of required proxy services, information about the remote service, client's preferences, etc. The proxy sends the retrieved request to the *PFS* to find a new proxy.

Step 2

After finding a new proxy, the *PFS* notifies the current proxy about the address of the new proxy.

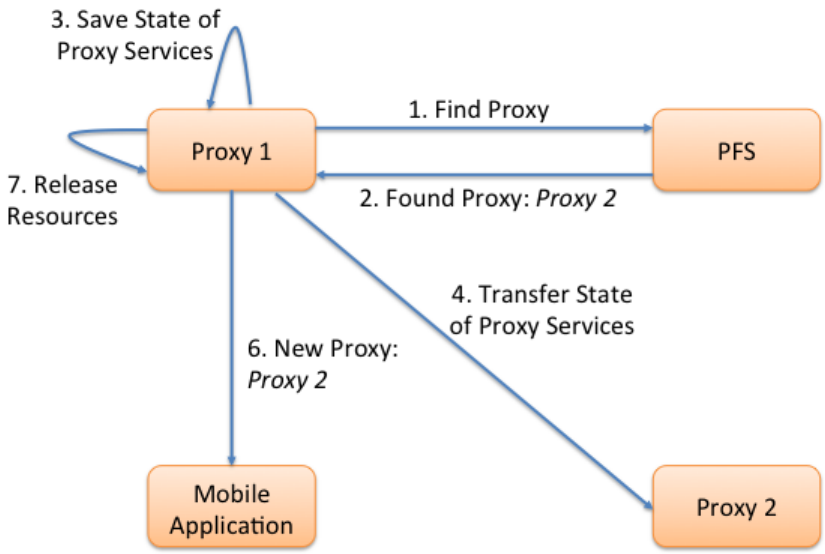


Figure 5.1: Handoff Operation

Step 3

The current proxy is responsible for saving the state of proxy services that support the client application. The information included in the state of a proxy service depends on the functionality of the proxy service. Every piece of information that is needed by a proxy service on the new proxy to provide the service continuation for the client application should be included as state information. As much as possible, the service continuation on the new proxy should be provided in a way that the client application is not forced to redo an already performed action, as much as possible. Realizing what kind of information should be included in the state information to meet the above mentioned goal is on the proxy service provider. The state transfer is descried in more detail in section 5.2.4.

Step 4

The saved state of proxy services and other data required by proxy services are transferred to the new proxy.

Step 5

After receiving the state files and other required data on the new proxy, the corresponding proxy services load their corresponding state and related data. The state loading might involve inserting data to databases, creating/copying required file, etc.

Step 6

The address of a new proxy, that is ready to service the mobile application, is sent to the mobile application by the old proxy.

Step 7

The old proxy frees all resources that were assigned to the mobile application and proxy services that where in use.

5.2.4 State of Proxy Services

The handoff operation assumes that a snapshot of the current state of each proxy service can be taken. State snapshots are specific to a proxy service. This section presents the state information that is preserved for several proxy services.

5.2.4.1 *Relay* Proxy Service

As described in section 3.3.1.2, the *Relay* proxy service is used to save the result returned from the remote service until it has validated that the mobile application has received the result successfully. For the handoff operation, the following tuple is stored by the *Relay* proxy service:

$$(r,c_i)$$

where r is the result from the remote service and c_j represents the client identifier of the mobile application that is to receive r. The handoff operation transfers the set of tuples that represent the result and the client identifier.

5.2.4.2 DataTransfer Proxy Service

The functionality of *DataTransfer* proxy service is described in section 3.3.1.2. The functionality of this proxy service includes transferring files from the mobile device to the proxy and handling disconnections. When a disconnection occurs, the *DataTransfer* proxy service stores the following tuple:

$$(n,f,c_i)$$

where n is the number of already transferred bytes from the mobile application, f is the path to the file that has been completely or partially transferred from the mobile application to the proxy, and c_i is the client identifier of the mobile application, transferring the file. During the handoff of a mobile application that uses the *DataTransfer* proxy service, the state of this proxy service is the set of tuples associated with the mobile application. There is one tuple for each transferred file. The transferred files that are indicated in the tuples with f should also be transferred to the new proxy. The transferred file might be needed for the proper operation of the mobile application. As a result, even if the file transmission has been done completely, there is still a need to transfer the file to the new proxy.

5.2.4.3 FileStreamer Proxy Service

The *FileStreamer* proxy service is described in section 3.3.2.2. The mobile application communicates with the *FileStreamer* proxy service over a socket connection to transfer files. The server socket belongs to the *FileStreamer* proxy service. After creation of the socket on the proxy, the *FileStreamer* proxy service stores the following tuple:

$$(p,c_i)$$

where p is the the port number of the socket server that the proxy listens on, and the c_i is the client identifier of the mobile application. As a result, the information that can be included to

the state of the *FileStreamer* proxy service during a handoff is the set of tuples associated with the mobile application.

5.2.4.4 CodeExec Proxy Service

As presented in section 3.3.1.2 the functionality of the *CodeExec* proxy service includes downloading the task executable from an online repository or the mobile device, installing the task and executing the task. After a handoff operation, the *CodeExec* proxy service should be able to set up the task on the new proxy without intervention of the mobile application. To be able to do this the instance of the *CodeExex* proxy service on the new proxy needs to know the download information of the task. This information can be transferred in the form of state to the *CodeExec* proxy service on the new proxy.

5.2.5 Common Tasks of Proxy Services

Saving the state information, transferring it to the new proxy and loading the state information on the new proxy is shown as steps 3-5 in figure 5.1. There are common activities that should be implemented by all proxy services regardless of the functionality of the proxy service.

In our prototype, an interface is defined that includes a method for each of the mentioned common tasks. All proxy services should implement methods of the interface. The design and implementation of the the interface methods is on the proxy service provider and is dependent on the functionality of the proxy service. It is possible to have a stateless proxy service. In this case, the interface implementation is empty.

5.2.6 Notifying the Mobile Application of Proxy Change

Handoff requires that client applications are notified of a change of proxy. The notification should minimize the disruption to the client application and be transparent to the mobile application user. This goal is met by including a common component in all mobile applications called the *Bridge*. The *Bridge* component maintains the address of the current proxy. All other parts of the client application, before calling any methods of any proxy services, should query the *Bridge* to find the proxy address. As a result, after a handoff operation is completed the *Bridge* component should be notified.

Immediately after the start of a handoff operation, the old proxy stops servicing the mobile application. The old proxy performs this by terminating all instances of proxy services that are servicing the mobile application. After finishing the setup of the new proxy with the state of required proxy services, the old proxy contacts the *Bridge* component of the mobile application

and provides it with the address of the new proxy. Hereafter, any call to proxy services from client application is directed to the new proxy by the *Bridge* component.

During the handoff operation, if the mobile application tries to access one of the proxy service on the old proxy, a failure happens since the proxy services instances are terminated. During the handoff operation all calls to the proxy services on the old proxy fail. In this case, the handoff looks similar to a disconnection to the proxy, and the client tries to re-call the proxy services. Each access requires a query to the *Bridge* to get the proxy address. Failures will continue to happen until the *Bridge* is updated.

5.3 Experiments and Results

The application used for the experiments in this section is the tourist assistance application, introduced in section 3.3.2. When the client part of this application on the mobile device needs to transfer a stream of images to the service part of the application on a remote host, the network latency plays an important role in the performance of the application. The amount of data transferred between the mobile application and the remote service will most likely cause a high number of disconnections. Thus this application requires the use of proxies to find a solution that handles the high network latency and disconnections.

5.3.1 Application and Used Proxy Service

As described in section 3.3.2, the client component of the tourist assistance application can use the proxy-based system for improving performance. As mentioned in section 3.3.2.2, the *CodeExec* proxy service can be employed to download the remote service of the tourist assistance application on a nearby proxy to reduce the communication delay. In this case, it is important to find a proxy with the smallest RTT to the mobile device. The proxy discovery mechanism can find the proxy with smallest network delay to the mobile device. This mean that the client application should include the RTT between the mobile device and the proxy as a factor in the proxy discovery request, and assigns a high weight to this factor.

The other proxy service that serves the client application is the *FileStreamer* proxy service. The *FileStreamer proxy service* is used to transfer a stream of images, where each image is in a file, from the client application to the proxy. The *FileSteamer* proxy service provides support for disconnection handling, i.e. *the FileStreamer* proxy service is aware of the possible disconnections and is able to re-connect and continue the communication between the client application and the service without the intervention of the client application or the service.

The goal of designing and executing the experiments presented in this section is to evaluate the impact of the handoff mechanism on the performance of the mobile application. The proposed handoff mechanism is supposed to maintain the desirable condition for the mobile application. In this case, the desirable condition for the client application means having a low network latency between the mobile device and the proxy during the use of proxy.

The mobile application used for the experiments of this section simulates the behaviour of the tourist assisting application by transferring a stream of files to the *FileStreamer* proxy service. In the experiments the effect of the handoff operation is studied only on the performance of the *FileStreamer* proxy service. This means that no remote service is included in the experiments, and consequently, no call to the *CodeExec* proxy service is made in the experiments. As a result, the components that play a role in the experiments are the followings:

- The mobile application that is implemented to use the *FileStreamer* proxy service to transfer a stream of files from the mobile device to the proxy.
- The *FileStreamer* proxy service that is used by the client application by receiving the files from mobile application.
- The proxy that provides the handoff operation.

5.3.2 Evaluation Metrics

The factor that is used to measure the performance of our experiments is the number of images transferred from mobile device to the proxy in 30 seconds.

5.3.3 Experimental Setup

The mobile device used for our experiments is an Acer Iconia Tab A500 with the Android 3.2. Two Linux servers are used as the proxies and one Linux server is used as the *PFS*. Apache Tomcat 7.0 is installed on all servers as the application server and the Apache Axis2 1.6 is used as the web service engine. The specification of the machines, wired links and wireless network are same as the those used for the previous experiments, mentioned in section 4.4.3. A schema of the components used in the experiments of this section is depicted in diagram 5.2.

The simulation of the RTT is done based on the description provided in section 4.4.7.2.

In the real world implementation of the system, the proxy, which is responsible for monitoring the RTT, periodically sends ping messages to the mobile device to get the value of RTT.

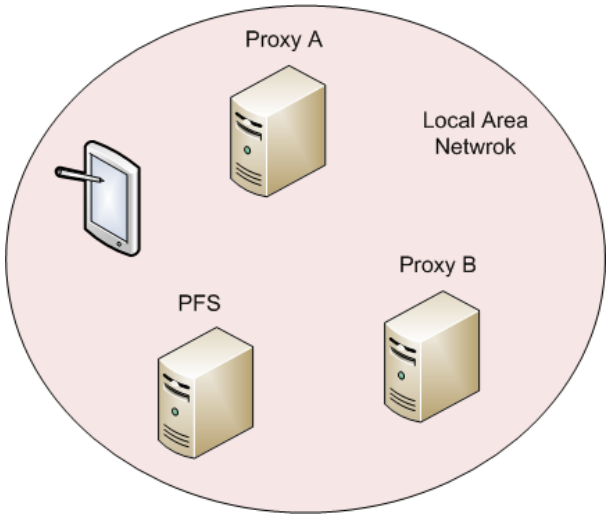


Figure 5.2: Experimental Setup for Handoff Experiments

In the implemented prototype of the system, RTT values are simulated at the mobile application and cannot be taken using the ping tool from the proxy. As a result, the mobile application periodically sends the current value of simulated RTT to the proxy.

5.3.4 Effect of Handoff Under Various Conditions

In these experiments we assume that an RTT value is defined that represents a threshold on the maximum RTT value accepted for the mobile application. This means that all RTT values below this threshold are acceptable for the mobile application, but an RTT value higher than the handoff threshold leads to the start of the handoff operation and consequently change of the proxy.

To measure the effectiveness of the handoff operation, the result obtained when a handoff occurs, is compared with the situation where no handoff happens. For this purpose, two scenarios are compared:

1. No handoff: In this scenario the handoff operation is not used regardless of the RTT value.

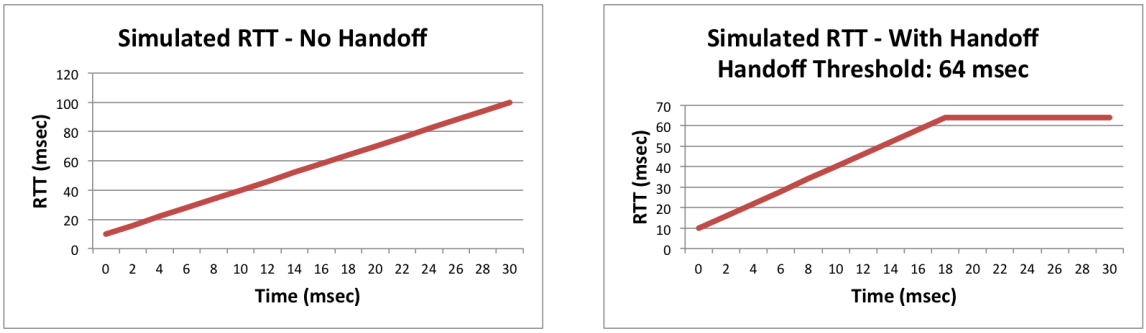
2. With handoff: A handoff operation is started when the RTT exceeds the handoff threshold. After the completion of the handoff operation, the client application transfers images to the new proxy.

We also consider two patterns of changes in RTT values. The first represents a linear increase in RTT and the other represents a spike increase in RTT.

5.3.4.1 Linear Increase in RTT

This experiment assumes that the RTT value increases gradually during the experiment as shown in figure 5.3a. The initial value of the RTT is 10 msec and linearly increases to 100 msec at the end of 30 seconds. This means that the RTT increases 3 msec every second. The RTT between the mobile device and the proxy in the *No Handoff* scenario changes according to the figure 5.3a.

With the *With Handoff* scenario, when the RTT exceeds the handoff threshold, and the mobile application is associated with a new proxy, it is assumed that the RTT value between the mobile device and the new proxy is equal to the handoff threshold and does not change until the completion of the experiment. For example, in the figure 5.3b the value of RTT is shown during the experiment when the handoff threshold is equal to 64 msec and a handoff has occurred. As shown in figure 5.3b, the RTT initially increases same as the *No Handoff* scenario, but at the 14th second of the experiment, the RTT exceeds the handoff threshold, i.e. 64 msec. Consequently, a handoff operation is executed and afterwards, until the end of the experiment, the RTT between the mobile device and the new proxy is equal to 64 msec. The handoff threshold is assigned to the RTT after handoff operation to simulate the worst case. In other cases the RTT between the mobile device and the new proxy should be lower than the handoff threshold, otherwise another handoff operation is started.



(a) No Handoff Scenario (b) With Handoff Scenario

Figure 5.3: Linear Increase of RTT

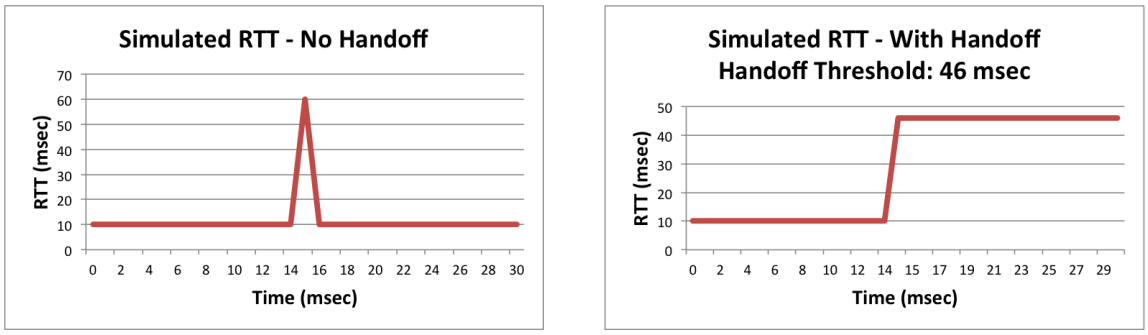
5.3.4.2 Spike Increase in RTT

In a real world environment, the RTT between two components fluctuates and might not change linearly. It is possible that the RTT increases for a short period of time and then returns to its previous level. To understand the effect of handoff started because of a spike increase in the RTT, we have designed and examined another set of experiments. In this scenario it is assumed

that the RTT value is equal to a constant value during the experiment, but increases to a value larger than the handoff threshold for a short period of time.

The simulated RTT values for the *No Handoff* experiments is shown in figure 5.4a. As shown in the figure, the RTT is equal to the constant value of 10 millisecond, but it increases from 10 to 60 millisecond for a length of 2 seconds in the middle of the experiments, from 14th second to the 16th second, then the RTT value returns to 10 milliseconds.

For the spike increase scenario, similar to the linear increase scenario, if the RTT exceeds the handoff threshold during the *With Handoff* experiments, a handoff operation is started and the RTT to the new proxy after the handoff completion is assumed to be equal to the handoff threshold. The simulated RTT for the the handoff threshold of 46 msec is shown in the figure 5.4b. During the spike increase in the RTT, the value of RTT has exceeded the handoff threshold, at almost the 15th second of the experiment and a handoff operation is executed. Afterwards, the RTT value between the mobile device and new proxy is equal to 46 msec, which is the handoff threshold.



(a) No Handoff Scenario

(b) With Handoff Scenario

Figure 5.4: Spike Increase in RTT

5.3.5 Results Assuming a Linear Increase in RTT

The threshold values used were 14.5, 19, 28, 37, 46, and 64 milliseconds and the file sizes used were 100 KB, 250 KB, 500 KB, 750 KB, 1 MB, and 1.5 MB. The file size does not change during an experiment.

The result of our experiments for the linear increase of RTT is depicted in figure 5.5. The calculated standard deviations are presented in table A.5 of appendix A. All data shown in this figure is calculated by averaging the results taken from three runs. The chart shows mostly the improvement of handoff cases over no handoff cases. The improvement is calculated as shown in equations 5.1.

$$\begin{aligned} \alpha &= Number\ of\ images\ transferred\ in\ \text{“}with\text{-}handoff\text{”}\ scenario \\ \beta &= Number\ of\ images\ transferred\ in\ \text{“}no\text{-}handoff\text{”}\ scenario \\ Improvement &= \frac{\alpha - \beta}{\beta} * 100 \end{aligned} \tag{5.1}$$

As shown in figure 5.5, the improvement for the largest file, 1.5 MB, is negative for all threshold values in contrast with the result for other smaller files. This result suggests that the handoff is not beneficial when the file is too large.

The other observation is that the improvement values are negative for the largest handoff threshold, i.e. threshold of 64 millisecond. It shows that it is better to start handoff on smaller values for the threshold. There is no significant difference between the result obtained for the file sizes of 100 KB, 250 KB, 500 KB and 750 KB for all thresholds but 64 millisecond. Almost all of them have a positive value between 0 and 20 percent.

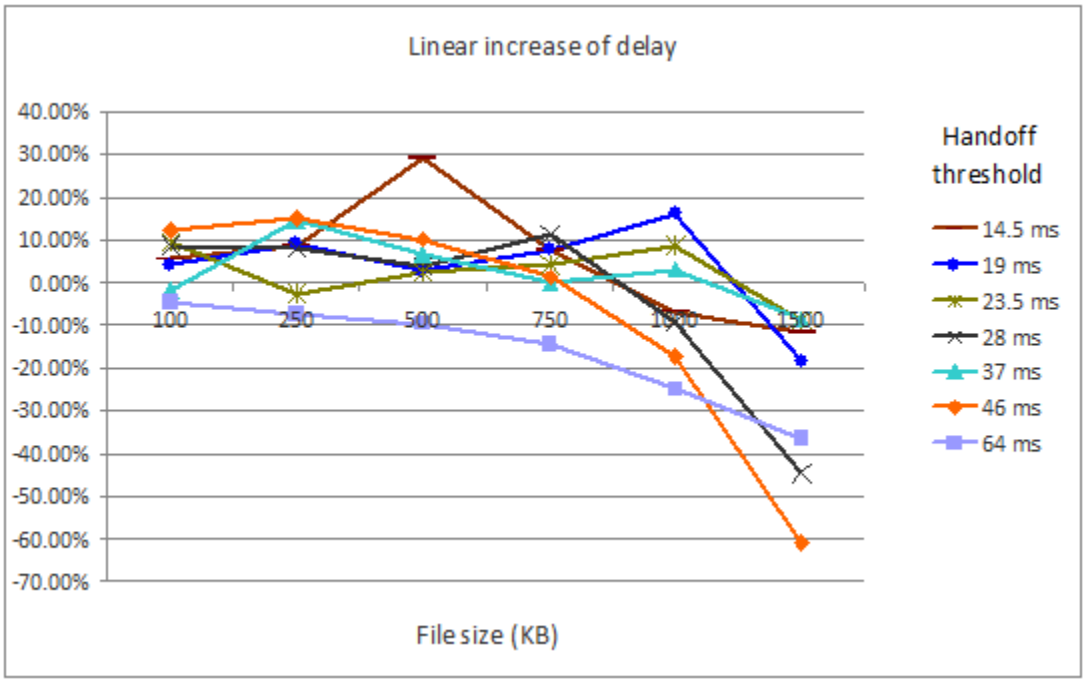


Figure 5.5: Linear increase of delay

The results suggest that handoff may not be effective for large files. The reason is that a handoff decision may be made while some files are in the process of being transmitted i.e., the handoff decision is made but before the mobile device can be informed of the decision there may be some files that have been transmitted. These files are lost and reflect a waste of time and bandwidth.

5.3.6 Results Assuming a Spike Increase in RTT

The handoff threshold values investigated for this scenario are 19, 28 and 46 millisecond. The file sizes examined are similar to the first scenario.

The result of experiments for this scenario is shown in the figure 5.6 and calculated standard deviations are presented in table A.6, appendix A. Each experiment is repeated three times and the result shown in the figure is the average of three repeats. As can be seen in the chart, almost all of improvement percentages for files larger than 250 KB are negative. This means that it has not been effective to do a handoff when a spike increase in the RTT happens. This suggests that it is necessary to have a mechanism in the system to realize the transient increases in the RTT and to not start the handoff process in those cases.

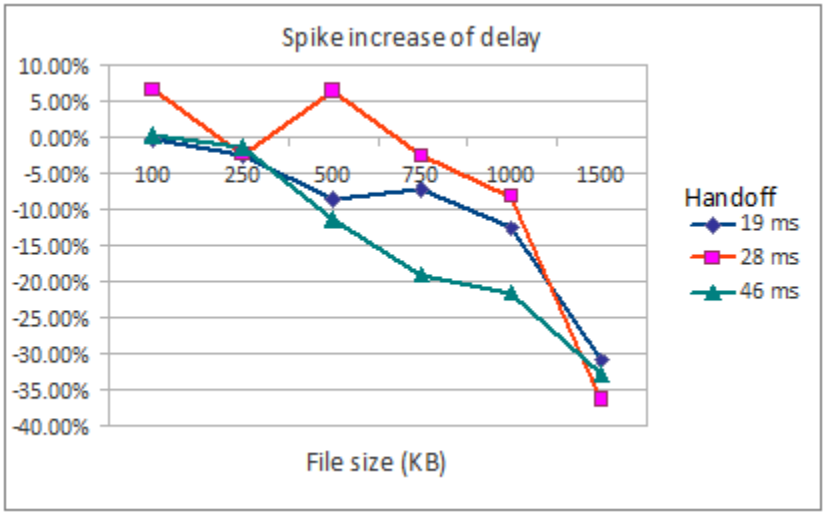


Figure 5.6: Spike increase of delay

5.3.7 Conclusion

The result of our experiments shows that the handoff mechanism provided in our proxy-based system is successful in keeping the communication delay between the client and service low. The application used in our experiments consecutively transfer files to the remote service. Remote service hosted on the proxy and is responsible for processing the photos and extracts extra information about the object. The result of our experiments shows that the handoff is effective and successful when file size is small, i.e. smaller than 750 KB. The results suggest that handoff operation is not efficient when transferring larger files. In addition, it is necessary to start handoff on smaller threshold values. The handoff process may result in worse performance when the threshold is exceeded because of a spike increase in delay. It is most likely that in

real world communications that RTT fluctuates. As a result, it is necessary to add mechanisms to distinguish a spike increase in delay from a list increase in the delay.

5.4 Summary

In this section the handoff mechanism provided in the proposed proxy-based mobile computing infrastructure was presented. The handoff mechanism aims to maintain the favourable condition for the mobile application. The favourable condition for the mobile device should be defined by the mobile application in the proxy discovery request, in the form of factors that are important for the functionality of the mobile application, along with the acceptable values for those factors. The proxy is responsible for monitoring the conditions and if the value of one of the factors, that are specified in the proxy discovery request of the mobile application, changes to something not favourable for the mobile application, triggers the handoff mechanism. During the handoff the task of servicing the mobile application is transferred to another proxy that is able to provide the acceptable service for the mobile application. This transfer to the new proxy is transparent from the mobile application. The handoff mechanism is done with the cooperation of the old proxy, *PFS*, and the new proxy. The experiments are designed to find the affect of the handoff mechanism on the performance of the mobile application. The example mobile application used for the experiments is the tourist assistance application and the important factor for the mobile application is the communication delay. The result of the experiments show that the handoff mechanism is able to provide better performance for the mobile application by keeping the communication delay below the threshold acceptable for the mobile application.

Chapter 6

Programming Model

The proxy services can either be used as a bridge between mobile applications and cloud services or can be used to enable mobile applications to offload computation. The distribution of computation that spans devices, proxy servers and clouds can complicate software development. This chapter focuses on a programming model that considers the use of proxies for both scenarios described above.

This chapter presents a programming model of the proxy-based system proposed in this work. A programming model provides a set of abstractions used by developers and runtime support that allows the abstractions to leverage the system for use by developers.

This chapter is organized as follows: Section 6.1 describes programming models related to mobile computing found in the literature. In section 6.2 the API that is provided for mobile application developers to implement applications that use the proposed proxy-based system is introduced. It is also shown how the provided API is used in the implementation of the example mobile applications used for the experiments, e.g., the remote printing and the tourist assistance applications.

6.1 Related Work

Programming models should facilitate application development. Abstractions are needed that represent objects to be manipulated without exposing the complexities in their implementation. This section describes some of the representative work in programming models for mobile application development that involves offloading and some form of proxy.

Dynamic partitioning allows for application partitions to be offloaded from the mobile device to a a proxy based on changes in the environment. Complexities added to the system include how to partition the application and how to realize offloadable partitions.

In *MAUI*, introduced by Cuervo et al. [23] (described in section 2.1.1.1), a dynamic partitioning model is provided. The partitioning granularity is at the method level. Migratable methods are annotated manually by the programmer. During the execution of the application, *MAUI* makes decisions about offloading migratable methods.

In the *Chroma* system [14] (section 2.1.1.9) the developer is responsible for determining different approaches to application partitioning. As described in section 2.1.1.9, the application programmer provides a tactic file, containing possible partitioning tactics, for the application. The tactic file includes the list of methods that can be called remotely as well as the data dependency between methods. The file also includes the order of executing methods, and if they can be called in parallel, or should be called sequentially. A problem with this approach is that if the application size increases, and there is a large number of remotely callable methods, it would be cumbersome for the developer to realize all possible tactics.

In Verbelen et al. [66] (section 2.2.4) the mobile application should be developed based on a component-based programming model [3]. The example AR application implemented in the paper is implemented as a set of OGSi [7] components. The *Execution Environment* entities, on the mobile device and the cloud node, are able to stop or start components, resolve components dependencies, and monitor the resource usage of components.

The *Cuckoo* framework [40] (section 2.1.1.4) provides a programming model that is integrated with Eclipse. While parts of the application development are automated, the application developer should provide two implementations for offloadable components of the application: one to be executed on the mobile phone and the other is to be executed on the proxy.

In the *CloneCloud* system, proposed by Chun et al. [20, 21] (section 2.2.1), no programming model is proposed for using the *CloneCloud* framework. Unmodified mobile applications can be executed on the system. Although, it is required that migration points be realized automatically through an off-line phase. This requires a static analyzer that is executed for each application before using the application. In the experiments presented in Chun et al. [21], the off-line static analyzer took on average 23 seconds for all example applications, while running on a desktop machine.

In almost all of the related work, the proposed programming model is heavily dependent on the cooperation of the application developer. The only work that does not require the assistance of application developer is the *Cuckoo* framework that does not working efficiently because of the time consuming initial offline phase. In addition, none of programming models discuss proxy discovery. All assume that the proxy already is known by the mobile application.

The aim of our programming model for our proposed system is reduce the burden of developers. If the mobile application developer decides to use generic proxy services to access a remote service, the developer only needs to realize which proxy services will be useful. If the

mobile application uses a proxy service it uses the API provided by the proxy services.

The task of proxy discovery is assigned to the *PFS*. Mobile applications can simply contact the *PFS* to find the most appropriate proxy. Additionally, proxies and the *PFS* should be able to carry out the handoff operation cooperatively.

6.2 Programming Model

The components of the proposed proxy-based system are the *PFS*, proxy, and proxy services. To enable proxy and proxy service providers to add their proxies and proxy services to the system, and also enable mobile application developers to create mobile applications using the system there is need to define APIs for each of the components. In this section the API that should be provided by components is presented.

6.2.1 *Bridge*

The *Bridge* component, previously introduced in the section 5.2, is created by the mobile application. A *Bridge* object maintains the following information for a proxy: IP address of the proxy, the client identifier used by the proxy to represent the mobile application, and the port number of the messaging server. The messaging server is used by proxy to notify mobile application about change of proxy because of a handoff. Operations on the *Bridge* object include assigning and retrieving these values. The Bridge also contains operations that are used to interact with other components. These are briefly described in this section and presented in table 6.1.

6.2.1.1 findProxy Operation

The *findProxy* operation is called by a mobile application that wants to use the proxy-based system. This method calls the *findProxy* operation of a *PFS* to find a proxy. The results received from *PFS* are used to set the proxy IP address and the client identifier of the *Bridge* object.

After receiving the IP address of the found proxy, the *findProxy* operation calls the *helloProxy* operation of the selected proxy to setup the proxy for serving this mobile application. The proxy, upon being contacted from the *findProxy* operation, installs the required proxy services and starts the messaging server process that will be used for sending messages from the proxy to the mobile application. The port number that the messaging server process listens on is returned by the *helloProxy* operation of the *proxy*. The value returned by the *findProxy* to the mobile application is a status code showing the success or failure of the operation.

Table 6.1: *Bridge* API

Status findProxy(ProxyFindRequest request)
ProxyFindRequest(ServiceInfo remoteService, ProxyServiceInfo[] proxyServices, Factor[] factors, Weight[] weights)
Status byeProxy()

6.2.1.2 *ProxyFindRequest*

The *ProxyFindRequest* component is used by the mobile applications to specify its preferences on the properties that the proxy should have.. As described earlier in section 4.2, the preferences of the mobile application includes the address of remote service, which is used for calculating the distance, a set of attributes and their weights. This component is passed as a parameter to the *findProxy* method. The *ProxyFindRequest* component should be created and filled before calling the *findProxy* method.

6.2.1.3 byeProxy Operation

This method is called by the mobile application after the mobile application is finished with the proxy and does not need to use proxy services anymore. The *byeProxy* operation calls the *byeProxy* method of the proxy API, by passing the client identifier of this mobile application. The value returned by the *byeProxy* to the mobile application is a status code showing the success or failure of the operation.

6.2.2 *PFS* API

The operations included in the *PFS* API are listed in table 6.2. The *PFS* allows mobile applications to find proxies. Proxies register with the *PFS*.

6.2.2.1 registerProxy Operation

The *registerProxy* operation is called by a proxy that intends to register at the *PFS*. The *ProxyInfo* object passed to it contains information about the proxy that is used during the proxy discovery operation. The information includes the IP address of the proxy and the list of available proxy services on the proxy. The value returned by the *registerProxy* to the proxy is a status code showing the success or failure of the operation.

Table 6.2: *PFS* API

Status registerProxy(ProxyInfo proxyInfo)
ProxyInfo(IPAddress address, ProxyServiceInfo[] proxyServices)
FoundProxyInfo findProxy(ProxyFindRequest request)

6.2.2.2 findProxy Operation

The *findProxy* operation of *PFS* can be called either by a mobile application that wants to use the proxy-based system, or a proxy that wants to handoff a client to another proxy.

This method contacts proxies by executing the algorithms presented in section 5.2 of chapter 5. Upon selecting a proxy, this operation returns the IP address of the proxy and the client identifier received from the proxy.

6.2.3 Proxy API

All the responsibilities of the proxy, including the tasks done during the proxy discovery and during the handoff are done without the intervention of the mobile application. The only method of the proxy that is accessed directly by the mobile application is the *byeProxy* method, shown in table 6.3 and described in the rest of this section.

6.2.3.1 isAvailable Operation

The *isAvailable* operation is called by the *findProxy* operation of a *PFS* by passing the request of a mobile application as parameter. This operation checks the condition of the proxy and the requirements of the mobile application, as indicated in the request, to determine if the proxy can accept this request. The decisions of the operation is based on the policies and goals of the proxy provider. If the *isAvailable* operation decides to accept the request, it generates a client identifier that is returned to the *PFS*; otherwise it returns a status code indicating that the request was rejected. If the request was accepted, the operation inserts the information about the client, i.e. the client's request and its assigned identifier, to its database.

6.2.3.2 helloProxy Operation

The *helloProxy* operation is called by the *Bridge* component of the mobile application after the proxy has accepted the client's request, and the *PFS* has chosen the proxy. The mobile application passes its client identifier that is assigned by the proxy and was previously received from *PFS*. The *helloProxy* operation retrieves the request of the mobile application, installs the

Table 6.3: *Proxy* API

IsAvailableResult isAvailable(ProxyFindRequest request)
PortNo helloProxy(ClientIdentifier clientID)
Status byeProxy(ClientIdentifier clientID)

required proxy services, and starts the messaging server. Finally, *helloProxy* returns the port number that the messaging server process listens on, to the mobile application.

6.2.3.3 byeProxy Operation

The *byeProxy* operation is called by mobile applications. The *byeProxy* operation terminates the operation of all the proxy services and the messaging server that was created for this mobile application. In addition, using the provided client identifier, the proxy removes all the data related to this mobile application from its local database. As a result, calling the *byeProxy* method causes the release of all resources assigned to this mobile application on the proxy. A status code is returned to the *byeProxy* method that indicates the success of the operation.

6.2.4 Proxy Services API

Four proxy services are realized and implemented for the experiments. The functionality of these proxy services is presented in the section 3.3. For each proxy service an API is provided that contains methods that allows the mobile application to use the functionality provided by the proxy service. The API of a proxy service consists of methods specific to the proxy service as well as methods common to all proxy services. The specific and common APIs of proxy services are introduced in this section.

6.2.4.1 *Relay* Proxy Service API

The methods provided for mobile applications in the *RelayPS* API are shown in the table 6.4. The description of the methods is described in the rest of this section.

useRelayPS Operation

The *useRelayPS* operation is used to call the *Relay* proxy service. The *RelayPSRequst* component is passed to the *useRelayPS* method as parameter. The *useRelayPS* operation calls the *Relay* proxy service that subsequently calls a method of the remote service as specified in the *RelayPSRequst*.

The *useRelayPS* operation calls the *Relay* proxy service repeatedly until a result is received from the *Relay* proxy service or the number of attempts exceeds some threshold value.

Table 6.4: *RelayPS* API

UseRelayResult useRelayPS(RelayPSRequest request)
RelayPSRequest(ServiceInfo remoteService, MethodInfo method, Param[] mehtodParams)
Status removeResult(RelayPSRequest request)

If a disconnection between the mobile application and the *Relay* proxy service occurs, the result of calling the remote service will be saved in the database of the *Relay* proxy service. The *useRelayPS* that has not received the result because of the disconnection, calls the *Relay* proxy service again. The *Relay* proxy service upon receiving the call, realizes that it is a call from a returning client, by checking the client identifier value, and retrieves the result from its database and returns the result to the *useRelayPS* method.

RelayPSRequest

The *RelayPSRequest* component includes the IP address of the remote service, the name of the remote service method that should be called, and the values that should be passed to the method as parameters.

removeResult

After successfully receiving the result, the mobile application uses the *removeResult* operation to notify the *Relay* proxy service that the result associated with the request, can be removed from its database. A status code in returned to *removeResult* method that indicates the success of the operation.

6.2.4.2 *DataTransfer* Proxy Service API

The only method available in the *DataTransferPS* API is shown in table 6.5. The description of this method is described in the rest of this section.

useDataTransferPS Operation

To transfer a file using the *DataTransfer* proxy service to the proxy, the mobile application uses the *useDataTransferPS* operation. The path of the file on the mobile device is passed as a parameter to the method. The *useDataTransferPS* contacts the *DataTransfer* proxy service to start a connection for file transmission on the proxy. After starting the connection, the *DataTransfer* proxy service returns the port number, that the connection listens on, to the *useDataTransferPS*. The *useDataTransferPS* operation starts a client thread that transfers the file using the established connection.

Table 6.5: *DataTransferPS* API

UseDataTransferPSResult useDataTransferPS(FileInfo fileName)

Table 6.6: *CodeExecPS* API

UseCodeExecPSResult useCodeExecPS(FileInfo taskFile, Argument[] arguments)

After the termination of file transmission the *useDataTransferPS* method contacts the *DataTransfer* proxy service and inquires it about the size of the transferred file. This is done to make sure that the file is transmitted successfully. If the termination of connection occurs as the result of an interruption, the transmitted file is not valid and the file size on the proxy will not be same as the file size on the mobile device. After receiving the size of file from the *DataTransfer* proxy service, it is compared with the size of the file on the mobile device. If the size of the file on the mobile device is larger than the size of the transmitted file, then this implies that the file transmission was interrupted before completing the file transmission. In that case, the *useDataTransferPS* contacts the *DataTransfer* proxy service to resume the file transmission and transfers the file contents that were not transmitted. Additional mechanisms can be added to the *DataTransfer* proxy service to check the integrity of the transferred file.

The path to the transmitted file on the proxy is returned by the *useDataTransferPS* method.

6.2.4.3 *CodeExec* Proxy Service API

The *CodeExecPS* API consists one operation as shown in the table 6.6. The description of *useCodeExecPS* is presented in this section.

useCodeExecPS Operation

The *useCodeExecPS* method can be used to run a task on the proxy. The *useCodeExecPS* is called by passing two parameters: the reference to the task file and the list of arguments that should be given to the task file as input. The reference to the task file is the path to the file on the proxy machine or the download URL of the task. If the reference to the task file is a URL, the *CodeExec* proxy service downloads the task, installs it, and executes the task by passing the arguments received from the *useCodeExecPS* method. If the task file is available on the mobile device, the *DataTransfer* proxy service can be used to transfer the task file to the proxy.

The result of the executed task is returned by the *CodeExec* proxy service.

Table 6.7: *FileStreamerPS* API

PortNo setup()
Status transferFile(FileInfo fileName)

6.2.4.4 *FileStreamer* Proxy Service API

The *FileStreamerPS* API is shown in the table 6.7. The description of the operations provided in the *FileStreamerPS* API is described in this section..

setup Operation

This operation should be called at the beginning of using the *FileStreamer* proxy service. The operation contacts the *FileStreamer* proxy service. This operation creates a connection specifically for the transmission of files through the creation of a server side socket. The *FileStreamer* proxy service returns the server port number to the *setup* operation.

transferFile Operation

The second operation, the *transferFile* uses the server socket created by the setup operation to transfer a file. The *transferFile* operation can be called for multiple files. The same server socket can be used for all file transmissions. There is no need for the mobile application to wait until the completion of a file transfer to send the next file. The mobile file can start the transmission of multiple files one after another without waiting for the completion of previous file transfers. A status code in returned to *transferFile* operation that indicates the success of the operation.

6.2.4.5 Common Handoff API

The handoff operation is designed to be started and proceed without the intervention of the mobile application. It is also important to keep the handoff process transparent to the mobile application developer. Operations of proxy services described in this section can be used for achieving this goal. It is possible that the API operations contact the proxy in the middle of a handoff operation.

The proxy services are designed to handle the situation where a mobile application contacts a proxy during a handoff. This situation is handled with the handoff operation that when called tags the mobile application as ”migrating”. During a handoff operation, when a request a received from the mobile application, a proxy service first checks the state of the mobile application at the proxy. If it is tagged as “migrating”, the proxy service returns a specific code to the mobile application. The calling method of the mobile application by receiving that

Table 6.8: Handoff API

Status handoff(ClientIdentifier clientID, ProxyInfo newProxy)
Status loadStatusFile(FileInfo statusFile)

specific returned value from the called proxy service finds that the proxy is in the middle of a handoff operation. The method waits briefly before calling the intended proxy service again. The length of the wait depends on the time it takes to complete the handoff process.

Proxy services should be able to cooperate with the proxy in collecting and transferring their status. This is required for the continuation of a proxy service activity on the new proxy. To make it possible all proxy services are required to provide a common API that includes the operations in table 6.8.

handoff Operation

This operation is called by the local proxy. The *handoff* operation is responsible for creating the status file of the proxy service and discovering other data that should be transferred to the new proxy. After creating the status file and collecting the information about other required data, as described in step 3 of handoff management process (section 5.2.3), it transfers the status file and other resources to the new proxy. The information that should be written in the status file and other resources that should be transferred are dependent on the functionality of the proxy service, as described in chapter 5, section 5.2.4. A status code in returned to *handoff* operation that indicates the success of the operation.

loadStatusFile Operation

This method is called by the old proxy. The parameter passed to it is the path to the status file that is already transferred to the new proxy. This method loads the information in the status file. The information in the status file and the way of loading it is dependent on the functionality of the proxy service, as described in section 5.2.4. A status code is returned to *loadStatusFile* operation that indicates the success of the operation.

6.2.5 How Mobile Applications Use the API

This section shows how mobile applications use the proxy services through the programming model presented in this section.

```
1 #include Common;
2
3 Bridge bridge = new Bridge();
4
5 ProxyFindRequest proxyFindRequest = new ProxyFindRequest(printerFinderAddress,
      ‘‘RelayPS’’, {‘‘PSs’’,‘‘RTT’’}, {0.5,0.5});
6
7 ProxyAddress = bridge.findProxy(proxyFindRequest);
```

Listing 6.1: Finding a Proxy to Call the Printer Finder Service

6.2.5.1 Finding a Proxy

The code presented in listing 6.1 shows how a mobile application can call the *Printer Finder* service through a proxy. In this example, the mobile application developer realizes that the *Relay* proxy service can be helpful in calling the *Printer Finder* service, as described in the section 3.3.1.

In line 3 an instance of the *Bridge* component is created. An instance of the *ProxyFind-Request* is created afterwards. The parameters passed to this method include:

- The address of the remote service, i.e. the *Printer Finder* service.
- The list of required proxy services which for this example is only the *Relay* proxy service.
- The list of attributes that are important for this mobile application. In this example, this includes the available proxy services, shown by “PSs”, and the RTT value, shown by “RTT”.
- The list of attributes’ weights, which in this example is 0.5 for both attributes.

After creating the *ProxyFindRequest*, the mobile application calls the *findProxy* method of the *Bridge* to find a proxy at line 7 and this method returns the address of the chosen proxy.

6.2.5.2 Calling Printer Finder through *Relay* proxy service

The code in listing 6.2 shows how the *Relay* proxy service on the proxy can be used to call the *Printer Finder* service by a mobile application using the provided API.

A new instance of the *RelayPS* is created in line 3. The *RelayPS* object encapsulates the data returned from the *Relay* proxy service operations and the operations needed to communicate with the *RelayPS* proxy service. The parameters that should be passed to the *Printer Finder* service are set in line 4. The parameters include the current location of the mobile device and the access level of the mobile device users, for example “student”, “professor”, or “guest”.

```
1 #include RelayPS;
2
3 RelayPS relayPS = new RelayPS();
4 Param[] parameters = {MyIPAddress, MyAccessLevel};
5
6 RelayPSRequest relayPSRequest = new RelayPSRequest(printerFinderAddress, "
      findPrinter", parameters);
7
8 String closestPrinter = relayPS.useRelayPS(relayPSRequest);
```

Listing 6.2: Calling Printer Finder through Relay Proxy Service

The current location of the mobile device can be the IP address on the mobile device or its GPS coordinate. The *Printer Finder* service is designed such that it determines the distance calculation to be used by checking the value of the first parameter. If the first parameter is an IP address the RTT is used as the distance measure. Otherwise if the first parameter is the GPS coordinate, the physical distance is calculated.

An instance of *RelayPSRequest* is created in line 6 by passing the required parameters. The created *RelayPSRequest* object is passed to the *useRelayPS* that calls the appropriate method of the *Printer Finder* service in line 8. The result of calling this method is the IP address of the closest printer, that is returned by the *useRelayPS*.

6.2.5.3 Calling *Printing Service* through *DataTransfer* and *CodeExec* Proxy Services

The code for calling the *Printing Service* is presented in listing 6.3. It is assumed that an appropriate proxy has already been discovered. The discovered proxy has two required proxy services: the *DataTransfer* and the *CodeExec* proxy services.

New instances of *DataTransfer* and *CodeExec* classes are created in lines 4 and 5. The created instance of *DataTransferPs* is used in lines 7 and 8 to transfer two files to the proxy. The first file is the task file which is the client of the *Printing Service* and the second file is the file that is to be printed. The *useDataTransferPS* method, used for file transmissions, returns the path to the transmitted file on the proxy machine.

The arguments that should be passed to the *CodeExec* proxy service are set in line 10. The first argument, *PathToPrintFile*, is the value returned by calling the *useDataTransferPS* method in line 8. The second argument is the address of the closest printer, found by calling the *Printer Finder* service.

The *CodeExec* proxy service is used in line 12 by calling the *useCodeExecPS* method. The first parameter passed is the path to task file on the proxy, taken at line 7. The second parameter, which is the list of arguments, is set in line 10. By calling the *useCodeExecPS* method, the task

```
1  #include DataTransferPS;
2  #include CodeExecPS;
3
4  DataTransferPS dataTransferPS = new DataTransferPS();
5  CodeExecPS codeExecPS = new CodeExecPS();
6
7  String pathToTaskFile = dataTransferPs.useDataTransferPS(pathToLocalTaskFile);

8  String pathToPrintFile = dataTransferPs.useDataTransferPS(pathToLocalPrintFile
       );
9
10 Argument[] arguments = {pathToPrintFile, closestPrinter};
11
12 String status = codeExecPS.useCodeExecPS(pathToTaskFile, arguments);
```

Listing 6.3: Calling Printing Service through DataTransfer and CodeExec Proxy Services

```
1  #include FileStreamerPS;
2
3  FileStreamerPS fileStreamerPS = new FileStreamerPS();
4  fileStreamerPS.setup();
5
6  while (elapsedTime < THIRTY_SECONDS) {
7         picture = NextPicture();
8         fileStreamerPS.transferFile(picture);
9  }
```

Listing 6.4: Calling AR Service through FileStreamer Proxy Service

file is executed on the proxy, which results in the transferring the print file to the remote printer and printing the file.

6.2.5.4 Calling AR Service through *FileStreamer* and *CodeExec* Proxy Services

As explained in section 3.3.2.2, the *FileStreamer proxy service* in needed to call the AR application designed for our experiments. The code presented in listing 6.4 shows how the *FileStreamer* proxy service is used by the mobile application.

In line 3 an instance of the *FileStreamerPS* is created. In the next line, the *setup* method of the created instance is called. The functionality of the *setup* method, as described in section 6.2.4.4, includes preparing the *FileStreamer* proxy service on the proxy for serving the new proxy.

In the while loop, lines 6 - 9 that repeats for 30 seconds, the mobile application takes a new picture, e.g., from the camera, and uses the *transferFile* method to transfer it to the proxy.

6.3 Summary

The programming model provided for the system, aims at making it relatively easier to use the proxy based system. In this chapter the programming models provided in other mobile computing systems are described. The responsibilities of the mobile application developers, when using those programming models are presented. Afterwards, the programming model for the proposed proxy-based mobile computing system is introduced. The mobile application developer using a proxy service only needs to have access to the API of that proxy service. The APIs of proxy services should hide the complexities of the functionalities of proxy services from the mobile application developer. The example applications, used throughout the work, i.e. *Remote Printing* and the *Tourist Assistant*, are used to show how the API of the example proxy services e.g., *Relay, DataTransfer, CodeExec* and *FileStreamer* can be used by the mobile application developer. These examples show how the mobile application development can be done.

Chapter 7

Conclusion and Future Work

A review on other research work in the area of mobile computing revealed the gaps that need to be filled in this area. Our proxy-based mobile computing infrastructure is targeted at easing the communication between a handheld mobile device and remote services by employing a middleware made of proxies. Proxies provide various services that support mobile applications in accessing remote resources.

The proxy discovery mechanism proposed for the system takes into consideration the preferences of mobile applications and proxies, in addition to the condition of the environment in selecting a proxy for a mobile application. The system is also aware of the changes in the condition of communication environment and the condition of proxy. The handoff mechanism introduced in this work changes the proxy associated with a mobile application in response to the changes in the environment.

A prototype for the system and several example mobile applications are provided to evaluate the effectiveness of the proposed infrastructure. The results of experiments show that our infrastructure is successful in improving the performance of mobile applications to access remote services with the help of various proxy services. In addition, the proxy discovery and handoff mechanisms, evaluated in several experiments, show an improvement in the performance of mobile applications. The results of experiments also indicate that using the proposed mobile computing infrastructure does not add a noticeable overhead to the system.

A programming model is also provided that allows mobile application developers to use the proposed proxy-based infrastructure to develop mobile applications, as well as service owners to develop proxy services that could be integrated in the system and enables mobile applications to use services of service providers.

7.1 Future Work

Several possible future work exist that can be categorized in two categories, introduced in below.

7.1.1 Improve Functionality of System

It is possible to enhance the available features of the proposed infrastructure.

Some of the common requirements of mobile applications are implemented in this work as proxy services. Currently the proxy services used in this work are seen in the literature. We can enhance the prototype with more services.

Currently, a small number of factors (i.e., RTT between mobile device and the proxy, available proxy services, and proxy CPU load) are used in the process of proxy discovery and making decision to start a handoff process. However, there are other factors e.g., the trustworthiness of proxies, that play important roles in the quality of service provided by a proxy and should be considered in the proxy discovery process.

In all the experiments, the evaluation of the system is measured when only one mobile application is using the system. In the real world, the proposed proxy-based system should service multiple mobile applications simultaneously. It is needed to study the performance of the proposed system in the presence of multiple mobile applications, using various proxy services.

In the proposed system the decision to start a handoff process is made by the use of thresholds values on some factors. If the threshold is violated the handoff is started. The decision making algorithm can be enriched with more detailed policies. For example, the handoff operation be triggered by taking into consideration the RTT value, as well as the size of files in transfer. The policies represent different strategies. Future work should investigate tradeoffs.

The IP address of mobile device is used in the proposed handoff mechanisms to measure the RTT to proxies. In this work it is assumed that the IP address of mobile device does not change while using the system. However, this assumption is not always valid since a mobile device moves and connects to new networks. The IP address may change as a result. There is a need to enhance the proposed handoff operation with a mechanism to notify the proxy a change of the mobile device IP address.

It has been tried to provide an easy to use programming model for the proposed system, as described in chapter 6. Although, it is required to evaluate the easiness of the proposed programming model by comparing it to other programming models, mentioned in section 6.1, by designing and performing experiments.

7.1.2 New Features

Beside improving the current features of the proposed system, as described above, the system can be enhanced with new features to improve its applicability.

A proxy that services multiples mobile applications may need to choose between mobile applications for handoff process, when a handoff is needed. Efficient algorithms should be designed and added to the functionality of proxy to help it in selecting the best candidate for the handoff process. For example, when the threshold of the proxy CPU load is violated, it is more beneficial to realize clients that produce more load on the CPU and handoff one of those client.

A useful feature to be added to the system is the ability of proxies in contacting each other and exchanging information. This ability enables the system to distribute the tasks involved in the handoff process between proxies and remove the need to the intervention of *PFS*. In this way the scalability of the system increases and a larger number of proxies can be employed in the system.

The concept of handoff is introduced in other areas of networking. For example, the handoff between access point in the Media Access control (MAC) Layer of wireless networks, or the handoff between base stations in cellular networks are extensively studied and various methods are proposed. It will be helpful to employ some aspects of the handoff methods in other areas in the handoff mechanism proposed in this work.

As mentioned before, the remote services, proxies, and *PFSs* can be provided by various corporations. For the sake of profit increase, the corporations might be willing to cooperate by servicing the clients of each other, or to enable their own clients to use the services of other corporations. Since each corporation might have its own access regulations and security concerns, there is a need to provide mechanisms to create an inter-corporation proxy-based system that takes into consideration the security and privacy concerns of the corporations.

Bibliography

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. [Online; accessed Feb. 2014].

[2] Apache Deltacloud. http://deltacloud.apache.org. [Online; accessed Feb. 2014].

[3] Component-based software engineering. http://en.wikipedia.org/wiki/Component-based_software_engineering. [Online; accessed Feb. 2014].

[4] Google Compute Engine. https://cloud.google.com/products/compute-engine. [Online; accessed Feb. 2014].

[5] Jini. http://en.wikipedia.org/wiki/Jini. [Online; accessed Feb. 2014].

[6] lookbusy – a synthetic load generator. http://www.devin.com/lookbusy/. [Online; accessed Jun. 2014].

[7] Open Grid Services Infrastructure (OGSI). http://en.wikipedia.org/wiki/OGSI. [Online; accessed Feb. 2014].

[8] OSI model. http://en.wikipedia.org/wiki/OSI_model. [Online; accessed Feb. 2014].

[9] Round-trip delay time. http://en.wikipedia.org/wiki/Round-trip_delay_time. [Online; accessed Feb. 2014].

[10] Salutation. http://salutation.org/. [Online; accessed Feb. 2014].

[11] Service Location Protocol (SLP). http://srvloc.sourceforge.net/. [Online; accessed Feb. 2014].

[12] Universal Plug and Play (UPnP). http://en.wikipedia.org/wiki/Universal_Plug_and_Play. [Online; accessed Feb. 2014].

[13] Windows Azure: Microsoft's Cloud Platform. http://www.windowsazure.com/en-us/. [Online; accessed Feb. 2014].

[14] P. Bahl, Richard R.Y. Han, L.E. Li, and M. Satyanarayanan. Advancing the state of mobile cloud computing. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, MCS '12, pages 21–28, New York, NY, USA, 2012. ACM.

[15] R.K. Balan, M., M. Satyanarayanan, S.Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 273–286, New York, NY, USA, 2003. ACM.

[16] P. Bellavista, M. Cinque, D. Cotroneo, and L. Foschini. Self-adaptive handoff management for mobile streaming continuity. *Network and Service Management, IEEE Transactions on*, 6(2):80–94, 2009.

[17] P. Bellavista, A. Corradi, and L. Foschini. Context-aware handoff middleware for transparent service continuity in wireless networks. *Pervasive and Mobile Computing*, 3(4):439 – 466, 2007.

[18] J. Carmigniani, B. Furht, M. Anisetti, P. Ceravolo, E. Damiani, and M. Ivkovic. Augmented reality technologies, systems and applications. *Multimedia Tools and Applications*, 51(1):341–377, 2011.

[19] M.R. Catherine and E.B. Edwin. A Survey on Recent Trends in Cloud Computing and its Application for Multimedia. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 2(1):pp–304, 2013.

[20] B. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *HotOS'09: Proceedings of the 12th conference on Hot topics in operating systems*, Berkeley, CA, USA, 2009. USENIX Association.

[21] B.G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[22] A. Corradi, L. Foschini, J. Povedano-Molina, and et al. DDS-enabled Cloud management support for fast task offloading. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 000067–000074, 2012.

[23] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *MobiSys'10*, pages 49–62, 2010.

[24] G. Eysenbach. CONSORT-EHEALTH: Improving and Standardizing Evaluation Reports of Web-based and Mobile Health Interventions. *Journal of Medical Internet Research*, 13(4), 2011.

[25] D. Fesehaye, Y. Gao, K. Nahrstedt, and G. Wang. Impact of Cloudlets on Interactive Mobile Cloud Applications. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 123–132, 2012.

[26] G. Fritz, C. Seifert, and L. Paletta. A Mobile Vision System for Urban Detection with Informative Local Descriptors. In *Proceedings of the Fourth IEEE International Conference on Computer Vision Systems*, ICVS '06, pages 30–, Washington, DC, USA, 2006. IEEE Computer Society.

[27] S. Gammeter, A. Gassmann, L. Bossard, T. Quack, and L. Van Gool. Server-side object recognition and client-side object tracking for mobile augmented reality. In *IEEE International Workshop on Mobile Vision (CVPR 2010)*, June 2010.

[28] B. Girod, V. Chandrasekhar, D. M. Chen, N. M. Cheung, R. Grzeszczuk, Y. Reznik, G. Takacs, S. S. Tsai, and R. Vedantham. Mobile Visual Search. *IEEE Signal Processing Magazine, Special Issue on Mobile Media Search*, 2010.

[29] I. Giurgiu, O. Riva, and G. Alonso. Dynamic software deployment from clouds to mobile devices. In *Middleware 2012*, pages 394–414. Springer, 2012.

[30] T. Guan, E. Zaluska, and D. De Roure. A Grid Service Infrastructure for Mobile Devices. In *Proceeding of Semantics, Knowledge and Grid, 2005, First International Conference on*, pages 42–42, Nov. 2005.

[31] T. Guan, E. Zaluska, and D. De Roure. Extending Pervasive Devices with the Semantic Grid: A Service Infrastructure Approach. In *Proceeding of Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*, page 113, September 2006.

[32] M. Hefeeda and C. H. Hsu. Design and evaluation of a testbed for mobile TV networks. *ACM Trans. Multimedia Comput. Commun. Appl.*, 8(1):3:1–3:23, feb 2012.

[33] B. Huang, C. Lin, and C. Lee. Mobile augmented reality based on cloud computing. In *Anti-Counterfeiting, Security and Identification (ASID), 2012 International Conference on*, pages 1–5, 2012.

[34] Y. Huang, S. Mohapatra, and N. Venkatasubramanian. An energy-efficient middleware for supporting multimedia services in mobile grid environments. In *Proceeding of Information Technology: Coding and Computing, 2005. International Conference on*, volume 2, pages 220– 225, April 2005.

[35] Y. Huang and N. Venkatasubramanian. Supporting Mobile Multimedia Services with Intermittently Available Grid Resources. In *Proceeding of High Performance Computing - HiPC 2003*, volume 2913, pages 238–247, November 2003.

[36] Y. Huang and N. Venkatasubramanian. Supporting mobile multimedia applications in MAPGrid. In *Proceedings of the 2007 international conference on Wireless communications and mobile computing*, 2007.

[37] Y. Huang and N. Venkatasubramanian. Mobile data overlay (MDO): A data placement paradigm for mobile applications. In *Mobile Data Management, 2008. MDM'08. 9th International Conference on*, pages 173–180. IEEE, 2008.

[38] Y. Huang, N. Venkatasubramanian, and Y. Wang. MAPGrid: A New Architecture for Empowering Mobile Data Placement in Grid Environments. In *Cluster Computing and the Grid, 2007. Seventh IEEE International Symposium on*, pages 725–730, May 2007.

[39] S. Islam and J.C. Gregoire. Network Edge Intelligence for the Emerging Next-Generation Internet. *Future Internet*, 2(4):603–623, 2010.

[40] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: A Computation Offloading-Framework for Smartphones. In *Proceedings of the Second International Conference on Mobile Computing, Applications, and Services*, 2010.

[41] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. The Smartphone and the Cloud: Power to the User. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 342–348. Springer Berlin Heidelberg, 2012.

[42] A. Khalaj, H. Lutfiyya, and M. Perry. The Proxy-based Mobile Grid. In *The Third International ICST Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, June-July 2010.

[43] A. Khalaj, H. Lutfiyya, and M. Perry. A Proxy-Based Mobile Computing Infrastructure. In *Mobile, Ubiquitous, and Intelligent Computing (MUSIC), 2012 Third FTRA International Conference on*, pages 21 –28, June 2012.

[44] D. Koavchev, Y. Cao, and R. Klamma. Mobile Multimedia Cloud Computing and the Web. In *Multimedia on the Web (MMWeb), 2011 Workshop on*, pages 21–26. IEEE, 2011.

[45] E. Koukoumidis, D. Lymberopoulos, K. Strauss, J. Liu, and D. Burger. Pocket cloudlets. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 171–184, New York, NY, USA, 2011. ACM.

[46] D. Kovachev, Y. Cao, and R. Klamma. Mobile cloud computing: a comparison of application models. *arXiv preprint arXiv:1107.4940*, 2011.

[47] K. Kumar and L. Yung-Hsiang. Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? *Computer*, 43(4):51 –56, april 2010.

[48] H. Liang, D. Huang, and D. Peng. On Economic Mobile Cloud Computing Model. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 329–341. Springer Berlin Heidelberg, 2012.

[49] E. Lupu, N. Dulay, M. Sloman, and et al. AMUSE: autonomic management of ubiquitous e-Health systems, 2008.

[50] P. K. McKinley, F. A. Samimi, J. K. Shapiro, and et al. Service Clouds: A Distributed Infrastructure for Constructing Autonomic Communication Services. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 341–348, 2006.

[51] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. DYNAMO: A Cross-Layer Framework for End-to-End QoS and Energy Optimization in Mobile Handheld Devices. *Selected Areas in Communications, IEEE Journal on*, 25(4):722–737, 2007.

[52] M. Nidd. Service discovery in DEAPspace. *Personal Communications, IEEE*, 8(4):39–45, Aug 2001.

[53] G. Papagiannakis, G. Singh, and N. Magnenat-Thalmann. A survey of mobile and wireless technologies for augmented reality systems. *Comput. Animat. Virtual Worlds*, 19(1):3–22, February 2008.

[54] E. Park, H. Shin, S. Jo Kim, and et al. Selective grid access for energy-aware mobile computing. In *Proceedings of Ubiquitous Intelligence and Computing*, page 798807, 2007.

[55] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[56] R. Rajachandrasekar, R. Nagarajan, G. Sridhar, and G. Sumathi. Job submission to grid using mobile device interface. In *Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 170 –174, dec. 2009.

[57] A. G. M. Rossetto, V. C. M. Borges, A. P. C. Silva, and et al. SuMMIT - A Framework for Coordinating Applications Execution in Mobile Grid Environment. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 129–136, 2007.

[58] F. Samimi, P. McKinley, S. Sadjadi, and et al. Service Clouds: Distributed Infrastructure for Adaptive Communication Services. *Network and Service Management, IEEE Transactions on*, 4(2):84–95, 2007.

[59] F. A. Samimi, P. K. Mckinley, and S. M. Sadjadi. Mobile Service Clouds: A self-managing infrastructure for autonomic mobile computing services. In *in Proceedings of the Second IEEE International Workshop on Self-Managed Networks, Systems and Services (SelfMan*, pages 130–141. SpringerVerlag, 2006.

[60] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8:14–23, 2009.

[61] C. Shin, H. Kim, C. Kang, Y. Jang, A. Choi, and W. Woo. Unified Context-Aware Augmented Reality Application Framework for User-Driven Tour Guides. In *Ubiquitous Virtual Reality (ISUVR), 2010 International Symposium on*, pages 52 –55, july 2010.

[62] M. Specht, S. Ternier, and W. Greller. Mobile Augmented Reality for Learning: A Case Study. *Journal of the Research Center for Educational Technology*, 7(1), 2011.

[63] C. Taylor and J. Pasquale. Towards a Proximal Resource-based Architecture to Support Augmented Reality Applications. In *Workshop on Cloud-Mobile Convergence for Virtual Reality*, Waltham, MA, USA, March 2010.

[64] T. Minh Trung, Y. Moon, C. Youn, et al. A gateway replication scheme for improving the reliability of mobile-to-grid services. In *e-Business Engineering, IEEE International Conference on*, pages 456–463, 2005.

[65] T. Verbelen, P. Simoens, F. Turck, and B. Dhoedt. Adaptive application configuration and distribution in mobile cloudlet middleware. In *Mobile Wireless Middleware, Operating Systems, and Applications*, volume 65 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 178–191. Springer Berlin Heidelberg, 2013.

[66] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt. Cloudlets: bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, MCS '12, pages 29–36, New York, NY, USA, 2012. ACM.

[67] X. Zhang, W. Jeon, S. Gibbs, and A. Kunjithapatham. Elastic HTML5: Workload Offloading Using Cloud-Based Web Workers and Storages for Mobile Devices. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 373–381. Springer Berlin Heidelberg, 2012.

[68] W. Zhu, C. Luo, J. Wang, and et.al. Multimedia Cloud Computing. *Signal Processing Magazine, IEEE*, 28(3):59–69, 2011.

Appendix A

Standard Deviations

Table A.1: Standard Deviations for table 4.1

File Size	Re-transmission Time for *RemotePrint-WithoutProxy* (msec)	Recovery Time for *RemotePrint-ViaProxy* (msec)
300 KB	11.50	12.87
600 KB	19.69	26.77
900 KB	4.09	25.13
3 MB	35.62	57.24
5 MB	69.36	88.11
10 MB	162.45	172.45
15 MB	182.37	254.71
20 MB	95.11	306.40
25 MB	97.86	424.69

Table A.2: Standard Deviations for table 4.2

File Size	Execution time of *RemotePrint-WithoutProxy* (sec)	Execution time of *RemotePrinte-ViaProxy* (sec)
300 KB	0.02	1.75
600 KB	0.04	1.78
900 KB	0.01	1.84
3 MB	0.07	2.26
5 MB	0.14	2.54
10 MB	0.32	3.57
15 MB	0.36	4.55
20 MB	0.19	5.20
25 MB	0.20	5.62

Table A.3: Standard Deviations for table 4.3

File Size	Execution time of *AdaptiveRe-motePrint* (sec)
300 KB	0.03
600 KB	0.07
900 KB	0.01
3 MB	0.06
5 MB	0.13
10 MB	0.16
15 MB	0.31
20 MB	0.77
25 MB	0.10

Table A.4: Standard Deviations for table 4.5

Proxy A CPU Load (%)	*Proxy B* CPU Load (%)	MobileMatrixMultiplication Runtime (sec)	Chosen Proxy
10	10	0.16	*Proxy A*
30	10	0.88	*Proxy A*
50	10	0.42	*Proxy A*
70	10	0.55	*Proxy A*
90	10	1.05	*Proxy A*

Table A.5: Standard Deviations for figure 5.5

	Handoff Threshold (msec)						
File Size (KB)	14.5	19	23.5	28	37	46	64
100	3.57%	2.49%	5.04%	7.71%	7.71%	6.62%	13.73%
250	2.95%	7.46%	10.61%	4.80%	9.19%	10.49%	8.00%
500	21.00%	2.68%	3.15%	4.23%	2.70%	16.26%	10.98%
750	3.25%	7.34%	5.67%	16.86%	7.88%	5.57%	15.16%
1000	9.29%	14.85%	3.42%	3.17%	17.26%	6.63%	10.99%
1500	23.29%	32.25%	26.65%	16.55%	53.72%	13.25%	9.80%

Table A.6: Standard Deviations for figure 5.6

	Handoff Threshold (msec)		
File Size (KB)	19	28	46
100	0.53%	7.77%	7.89%
250	2.62%	8.47%	3.50%
500	1.26%	14.08%	8.36%
750	3.96%	12.00%	13.65%
1000	1.90%	13.18%	13.51%
1500	8.95%	2.29%	4.16%

Curriculum Vitae

Name:	Azade Khalaj
Post-Secondary Education and Degrees:	The University of Western Ontario London, ON 2007 - 2014 Ph.D. Computer Science University of Tehran Tehran, Iran 2002 - 2005 M.Sc. Software Engineering University of Tehran Tehran, Iran 1997 - 2001 B.Sc. Applied Mathematics
Related Work Experience:	Analyst/Programmer SHARCNET, The University of Western Ontario 2012 - 2014 University Level Teaching The University of Western Ontario June - July 2013 Teaching Assistant The University of Western Ontario 2007 - 2011

Publications:

- Azade Khalaj, Hanan Lutfiyya, "Handoff Between Proxies in the Proxy-Based Mobile Computing System," MOBILe Wireless MiddleWARE, Operating Systems and Applications (Mobilware), 2013 International Conference on, Bologna, Italy, November 11-13, 2013

- Azade Khalaj, Hanan Lutfiyya, Mark Perry, "A Proxy-Based Mobile Computing Infrastructure," Mobile, Ubiquitous, and Intelligent Computing (MUSIC), 2012 Third FTRA International Conference on, Vancouver, Canada, June 26-28, 2012

- Azade Khalaj, Hanan Lutfiyya, Mark Perry, "The Proxy-based Mobile Grid," The Third International ICST Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (Mobileware), Chicago, USA, June 30-July 2, 2010