DoD Mobile Client -
A Comparison between J2ME and Symbian Platforms

Sanjay Rajwani

Master of Science Thesis
Stockholm, Sweden 2012
DoD Mobile Client
A Comparison between J2ME and Symbian Platforms

Master Thesis

Sanjay Rajwani
<rajwani@kth.se>
2012-10-09

Examiner
Peter Sjödin

Industrial Supervisor
Jörgen Steijer, OptiCall AB

School of Information and Communication Technology
Royal Institute of Technology
ABSTRACT

With increasing need for mobile communication in today's world, there has been a constant hike in the cost of mobile communication as well. To cater this problem of increased cost of communication, OptiCall AB has devised an integrated solution called Dial over Data, or DoD. DoD is an integrated Client-Server solution combining a mobile interface with Asterisk's PBX and a GSM gateway in an intelligent way giving the customer various flexible ways of using their mobile phones and providing the ability to control their phones bills to a huge extent.

The client part of the DoD solution is developed over J2ME platform and runs over Java enabled mobile phones. Although the client application provides the basic functionalities of dialing a call, call records, contacts etc, it lacks user friendliness. In order to make a call, the application needs to be started from the mobile phones’ application folder which is not the most natural way of making calls from mobile phones. This thesis project aims at exploring the possibilities of and developing a client application which is transparent to the users of mobile phones and provides them with the natural way of making call using the DoD solution.

This thesis project analyses the existing J2ME client application for DoD solution to see how well it has been developed. It then compares the J2ME and the Symbian platforms to see which of the two platforms provides the capability of developing a transparent mobile client. In the end, we will develop a mobile client application which fulfils the major aim of this thesis work using one of the two platforms.
ACKNOWLEDGEMENTS

First of all, I would like to thank OptiCall AB for giving me this exciting opportunity of working on an interesting project. I would specially like to thank my supervisor at OptiCall AB, Jorgen Steijer, for giving me all the support that I needed during my stay at OptiCall. I would like to thank my thesis examiner at KTH, Peter Sjodin, for agreeing on examining my thesis work and also guiding me on how to work on thesis projects.

Throughout the project, I had received immense help from people at Forum Nokia. So I would like to thank Nokia for creating such a useful discussion forum and all the people, especially Mr. Sunil Kumar, a Forum Nokia Champion and Mr. Maldar Mohammad Mateen, a Forum Nokia Member, who responded to my queries. I would also like to thank Mr. Roy Sandgren, a Software Engineer at Densitet, who helped me a lot in achieving my major project goals.

Lastly, I would like to appreciate all the support I had received from my parents and my girlfriend. Thank you both for keeping up to my mood swings during that time.
# TABLE OF CONTENTS

Abstract ................................................................................................................................................................................3  
Acknowledgements .........................................................................................................................................................4  
List of Figures .....................................................................................................................................................................7  
1) Introduction .....................................................................................................................................................................8  
   General Overview ........................................................................................................................................................8  
   Problem Statement .....................................................................................................................................................8  
   Objective ..........................................................................................................................................................................9  
   Report Organization ...................................................................................................................................................9  
2) Background ...................................................................................................................................................................10  
   J2ME Application Development ......................................................................................................................................10  
      The Java 2 Platform ................................................................................................................................................10  
      The J2ME Platform ................................................................................................................................................11  
      Configurations and Profiles ........................................................................................................................................12  
      Mobile Information Device Profile and its GUI APIs ..........................................................................................14  
      MIDlet and its Life-Cycle .......................................................................................................................................17  
   Symbian Application Development ..........................................................................................................................17  
      Symbian OS .............................................................................................................................................................17  
      Symbian Application Framework ..........................................................................................................................19  
      UI Design Platforms ............................................................................................................................................19  
      View Architecture ................................................................................................................................................20  
   DoD Solution ...............................................................................................................................................................21  
      Components of DoD Architecture ........................................................................................................................21  
      DoD Architecture Description ................................................................................................................................22  
      DoD Solution Operation .........................................................................................................................................22  
3) Existing DoD Client ..........................................................................................................................................................24  
   Analysis: J2ME DoD Client ...........................................................................................................................................24  
      Class diagram .........................................................................................................................................................24  
      Important classes ..................................................................................................................................................25  
      Application usage ................................................................................................................................................26  
   Shortcomings And Requirements For DoD Client .......................................................................................................26  
      User Friendliness ................................................................................................................................................26  
      Application Design ...............................................................................................................................................27  
      Application Performance .......................................................................................................................................27  
4) Capability Analysis: J2ME vs. Symbian C++ ................................................................................................................28
LIST OF FIGURES

Figure 2.1 Java 2 Platform editions and their target markets [2] ................................................................. 11
Figure 2.2 Software Layers in a J2ME device [2] .......................................................................................... 12
Figure 2.3 CDC and its profiles [6] ............................................................................................................... 13
Figure 2.4 Relationships between J2ME configurations and J2SE [2] ..................................................... 14
Figure 2.5 Mobile Information Device Profile [2] ....................................................................................... 15
Figure 2.6 MIDP GUI Classes [8] .................................................................................................................. 15
Figure 2.7 Life-cycle of a MIDlet [8] ............................................................................................................. 17
Figure 2.8 Symbian OS layered system model [12] ..................................................................................... 18
Figure 2.9 UIKON and its underlying frameworks ................................................................................... 19
Figure 2.10 Class Derivation of S60 platform and S60 Symbian application [12] .................................. 20
Figure 2.11 Derivation of S60 view [27] .................................................................................................... 21
Figure 2.12 Architecture and Operation of DoD Solution [40] ................................................................. 22
Figure 3.1 Class Diagram of Java DoD Client ............................................................................................. 24
Figure 3.2 Java DoD Client Application GUI ............................................................................................ 26
Figure 4.1 The Interface Hierarchy of the Generic Connection Framework [18] ..................................... 31
Figure 4.2 Creating a HTTP session [34] .................................................................................................. 32
Figure 4.3 Creating and submitting a HTTP transaction [34] ................................................................... 33
Figure 5.1 Sequence diagram for Symbian DoD Client ............................................................................. 37
Figure 5.2 Class diagram for Symbian DoD Client Application ................................................................. 39
Figure 5.3 Symbian DoD Client Application start screen ........................................................................ 44
Figure 5.4 Setting options of Symbian DoD Client .................................................................................. 44
Figure 5.5 Symbian Client Server Settings .............................................................................................. 45
Figure 5.6 Symbian Client Call Back Settings ......................................................................................... 45
Figure 5.7 Symbian Client Call Through Settings .................................................................................... 45
Figure 5.8 Symbian Client Operational Settings ....................................................................................... 45
Figure 5.9 Symbian Client Default Call Method Setting ......................................................................... 46
Figure 5.10 Symbian Client White List ..................................................................................................... 46
Figure 5.11 Symbian Client Transparent Mode 1 .................................................................................... 46
Figure 5.12 Symbian Client Transparent Mode 2 .................................................................................... 46
Figure 6.1 User prompt to choose call method when it is set to Always Ask ........................................... 48
1) INTRODUCTION

GENERAL OVERVIEW

Among all other most essential ingredients of daily life is communication. A man has to communicate to convey his thoughts and ideas, to carry out his everyday business and to stay in touch with his friends and family. In a nutshell, a man has to communicate to improve the quality of life. As Anthony Robin says, “The way we communicate with others and with ourselves ultimately determines the quality of our lives.”

Communication has taken different forms since the ancient times. It has evolved from stone carvings and sending of written messages through messengers to different kinds of telephones in the recent times. And today, the most common form of communication is mobile communication. Mobile phones are so much used these days that this form of communication has now become the backbone of the society. Everyone, from businessmen to laymen and from old aged to school going children is using mobile phones to communicate with others. And, as the trends predict, its usage will raise even more in the near future. In fact, it is expected that mobile phones, along with its primary objective of enabling communication, will also be used for day to day businesses and tasks.

Witnessing so much increase in mobile communication, the mobile operators are becoming greedier. They are coming up with such intelligently designed and well thought mobile packages that the subscribers just cannot resist from purchasing them thinking that its even cheaper than the one they are currently using. The general rule is that cost of calls to fixed lines and to mobile subscribers belonging to operator other than that of callers’ is always very high.

The operators are charging even more from subscribers making calls from abroad using their local SIM cards, which is roaming subscriber calls. The roaming call costs are so high that individuals and companies are constantly in search for solutions/software which could reduce their local and roaming call costs while keeping ease of communication same as it is with normal usage of mobile phones.

Dial over Data (DoD) is one such solution/software invented by Opticall AB to reduce call costs of local and roaming calls. DoD is an integrated Client-Server solution combining a mobile interface with Asterisks PBX and a GSM gateway in an intelligent way giving the customer companies various flexible ways of using their mobile phones and providing the ability to control their phones bills to a huge extent.

This master thesis is focusing on the improvement and development of the client side of DoD. The client part of the solution is required to be a mobile client to enhance the mobility of the solution. The specific requirements of the mobile client are briefed in the following sections.

PROBLEM STATEMENT

As mentioned earlier, DoD is a Client-Server solution. The server part of DoD was designed and implemented by Ning Zhou [40] at Opticall AB as her master thesis work. Later on, Max Weltz [39] took the responsibility of improving the server part as his master thesis project. Max, along with improving the server part, developed a mobile client, which was programmed in Java Micro Edition, for testing the DoD solution with a mobile interface.
However, since Maxs’ master thesis work was more focused on stabilizing and commercializing the DoD server part, the java mobile client lacked user-friendliness, ease of use and many other such features which are essential for a good mobile application. And so, the DoD project is missing a decent mobile client application which can be of commercial use.

**OBJECTIVE**

The objective of this master thesis work is twofold. The first part of it deals with analyzing the existing DoD client, which was originally designed by Max Weltz [39]. This analysis will lead us to figure out the major shortcomings of the existing DoD client and from that we will deduce the specific requirements for the DoD client. We will then compare the capabilities of the J2ME and the Symbian platforms on the basis of those requirements to see which of the two platforms is better able to meet the needs of the new DoD client.

Once that is done, the second part of this thesis work is to implement a mobile client for DoD on the selected platform. So, the primary objective of thesis work is to implement a mobile interface for DoD which could either be implemented in J2ME or developed on the Symbian platform.

**REPORT ORGANIZATION**

Following this brief introduction, the rest of this report is organized as follows:

The next chapter explains the work that has already been done in the area of DoD and present the basics of the technologies that are either reviewed or used in this project to achieve the goals of this project.

Chapter 3 presents the analysis of the existing J2ME DoD client in terms of application architecture, application GUI and the operation of the application. From this analysis, the most important shortcomings and the requirements of the existing DoD client are figured out.

In the fourth chapter a comparison between the J2ME and Symbian platforms in terms of their capabilities of achieving the goals of the project. The chapter then ends by choosing one of the two platforms for developing the DoD client based on the capabilities analysis.

The fifth chapter presents the solutions and discusses the implementation details of the Symbian mobile client for DoD. It briefly explains the development environment and the application architecture followed by the features included in the application and a small guide on how to operate the application.

Chapter 6 compares the newly implemented Symbian DoD client with the existing J2ME DoD Client in terms of ease of use, user friendliness, speed of application execution etc.

The final chapter provides the conclusions derived from this thesis work and presents some recommendations for future work.
2) **BACKGROUND**

This chapter will give the reader an overview of the work done in the past and the technologies used in this project, which is needed for him/her to understand the later chapters. We will first explain the basic concept behind DoD, the architecture of the solution and its components and functionalities. Later on we will brief the user on the two platforms/technologies, Symbian and Java Micro Edition, that are thoroughly reviewed and used in this project. If not thoroughly, these two platforms will be discussed in enough depth to make the reader understand the following chapters.

**J2ME APPLICATION DEVELOPMENT**

In this section of the chapter, we will briefly explain Java and its three platforms, namely J2SE, J2EE and J2ME. Further into this section, we will look into the J2ME platform in more details, explaining its configurations (CLDC and CDC), its profiles (MIDP and others), its development environment and the basics of a MIDP GUI application. However, this section will not provide any details or code samples for MIDP applications. This is assumed that the reader is familiar with the basics of programming. Also note that we will not talk about J2SE and J2EE after the next section, since it is out of the scope of this thesis work.

**THE JAVA 2 PLATFORM**

Java is a Sun Microsystems invented programming language, released in the year 1995. Much of its syntax and programming concepts are derived from C++, but it has a much simpler object model than C++. The typical result of the compilation of Java code is Java bytecode. This Java bytecode is only readable by a Java Virtual Machine (JVM), which itself is a set of computer software programs that can run on any computer regardless of its architecture and can only read the Java bytecode. So, any Java code once compiled to Java bytecode can run on any computer which has a JVM running on it. The importance of JVM in the Java 2 Platform explains its most prominent feature, "portability".

Cross-platform compatibility being the most essential notion of Java, it is important to realize that this has its own limitations as well. Portability is only possible among similar hardware and operation systems. To bridge this gap, Java today is partitioned into three editions; the **Java2 Standard Edition (J2SE)**, the **Java2 Enterprise Edition (J2EE)** and the **Java2 Micro Edition (J2ME)**.

**J2SE** combines the core functionalities or the minimum support required for any Java platform. That is, it defines the JVMs and libraries which run on standard PCs and workstations.

**J2EE** is built on top of J2SE and enhances it by combining more advances features of the Java platform like Enterprise JavaBeans, JavaServlets API and JavaServer pages. Unlike J2SE, it is meant for building and developing huge, multi-tier and complex enterprise applications.

Unlike J2SE and J2EE which are platforms for servers, standard PCs and workstations, **J2ME** is a platform specifically for embedded OS platforms like home appliances, security systems, automotives and handheld devices like mobile phones and PDAs. J2ME is precisely a subset of the Java platform consisting of condensed collection of Java APIs for the development of software for the mentioned categories of devices.
Figure 2.1 below shows the market the Java 2 platform editions cater to. The J2ME part of the figure gives a lot more information which is discussed later in this chapter.

**THE J2ME PLATFORM**

Having said in the previous section, J2ME supports development of applications for small embedded OS devices. However, there is such a huge variety of devices within the set of small embedded OS devices, each having varying capabilities, like memory, graphical user interface (GUI) and networking, that it is hardly possible to build an application that would run on all the devices. For example, an application built for an automated washing machine will, by no means, be able to run on a mobile phone because of the difference in their capabilities. Therefore, the J2ME platform divides the complete set of devices in two categories, namely high-end consumer devices and low-end consumer devices.

The high-end consumer devices have considerably more user interface options and higher total memory than the low-end devices. The former usually have high bandwidth network connections using TCP/IP. Whereas the later has low bandwidth network connections which are not often based on TCP/IP. The examples of high-end consumer devices include wireless communicators, internet TV’s and automobile entertainment/navigation systems.

To cater the varying needs of the two categories of devices and to support portability, J2ME is divided into several components known as *configurations, profiles* and *optional packages*. A configuration and a profile are combined to provide an environment for developing applications for a particular family of devices.
A J2ME configuration is a specification of the Java Virtual Machine, the Java language features and a minimum set of class APIs needed for developing an application for a particular family of devices. The device manufacturers shall incorporate these specifications into their devices so that the application developers know what capabilities a particular device will have and can create as device-independent applications as possible. A device having, say similar memory, processing power, network connections etc, will have the same configuration.

A J2ME profile is layered on top of and extends a configuration by adding more APIs and classes which are essential for a subset of a particular family of devices. The most common examples of API added by a profile are persistent storage, network connectivity, security and graphical user interface (GUI). One or more profiles can be present on top of a configuration in a specific device.

The optional packages are APIs which can be used along with the configuration and profiles based on the needs of the application. Some commonly used APIs from optional package are J2ME Web Services API, Location API, Mobile 3D Graphics etc.

All these three components of the J2ME platform will run on devices’ host operating system. The Figure 2.2 below shows the relationship between the J2ME components and the host operating system.

**CONFIGURATIONS AND PROFILES**

Currently, the J2ME platform has two configurations known as the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC).

The Connected Device Configuration targets the high-end consumer devices, precisely having a minimum of 2 MB memory, 32-bit processors and good network connectivity. Since the CDC devices have reasonable memory and relatively good network connectivity, it is capable of supporting a compact version of the Java Virtual Machine (JVM), called the CVM or “Compact” Virtual Machine. The only difference between CVM and JVM is that CVM is able to run on small
memory. The core libraries supported by CDC are mainly inherited from J2SE and the remaining are custom-made specifically to suit the needs of the devices. Since most libraries in CDC are included from J2SE, the development environment for CDC-based applications is very much similar to that of desktop PCs and workstation applications.

The most common profile that extends the CDC is known as the **Foundation Profile**. This profile widens the CDC by providing support for classes that are absent in the CDC core libraries package and some advanced APIs from the J2SE which are not present in the CDC core libraries. The Foundation Profile, however, doesn't include graphical user interface classes. These classes are instead provided in the **Personal Basis** and **Personal** profiles. These profiles are layered on top of the Foundation Profile.

There are some more profiles that extend the Connected Device Configuration, but we are not going to talk about them as they are beyond the scope of this thesis work. Figure 2.3 below, however, shows the CDC and all its profiles.

![Figure 2.3 CDC and its profiles](image)

The **Connected Limited Device Configuration (CLDC)** is supported on devices which are smaller than the CDC supported devices that are the low-end consumer devices. These devices typically have memory budget as less as 160 KB and as much as 512 KB. The RAM sizes normally vary between 32 and 64 KB, where as the network connectivity is very intermittent and not so fast.

Due to the limited capabilities of the devices at which CLDC is targeted, it is practically impossible to support complete JVM features or Java core libraries. Hence, to support the Java runtime environment on such limited devices, CLDC condenses the VM and core libraries requirements, to allow portability on maximum number of devices. Therefore the VM for CLDC is a very trimmed down version of the JVM, with very limited capabilities that are just enough to satisfy the needs of CLDC. The VM for CLDC is called Kilobyte VM or KVM because of its reduced features and size. Also, the class libraries defined to be in the CLDC specification is a minimum subset of the J2SE libraries. These libraries are very basic and ignore the fact that a device has a
user interface or a keyboard for input. And so, these libraries, as well, satisfy the minimum requirements of CLDC so that they can address the needs of maximum possible devices.

It is important to note that applications for embedded OS devices need a subset of the features and capabilities required for applications that are meant to run on desktop computers. And so, the J2ME configurations combine the basic and core features of the Java 2 Standard Edition that are needed for these low profile devices. Figure 2.4 below shows the relationship between the J2ME configurations and J2SE.

![Figure 2.4 Relationships between J2ME configurations and J2SE](image-url)

Since the CLDC comes with such limited number of core libraries, it is barely possible for application developers to write applications without having any APIs for display, user interaction or persistent storage. In order to facilitate this, the Mobile Information Device Profile, or MIDP was defined to work on top of and extend the CLDC. It is mainly targeted for mobile phones and pagers. For, PDAs the profile is called PDA Profile, or PDAP.

In the next section, we will only discuss the details of MIDP and application development over MIDP, as this is directly related to this thesis work.

**Mobile Information Device Profile and its GUI APIs**

As mentioned earlier, it is nearly impossible for application programmers to write applications for mobile devices with the APIs provided in CLDC which do not have any support for user interaction, user interface, networking and storage. Mobile Information Device Profile (MIDP) is a J2ME profile which extends the core set of APIs provided by CLDC and provides the application programmers with the possibility of developing applications with advanced features. An application written using the APIs provided by MIDP and the APIs inherited by MIDP from CLDC is called a MIDlet. The next section will elaborate more on MIDlets.

Figure 2.5 below shows the logical position of MIDP and MIDlet within the software architecture of the device that implements it.
MIDP contains a considerably large set of classes for user interface and user interaction. The MIDP user interface library is implemented in javax.microedition.lcudI and the figure 2.6 below is showing a high level class diagram of how these classes are structured.

As shown in the diagram above, each MIDlet has a one-to-one relationship with the Display class. This illustrates that fact that each GUI application has a single instance of this class which represents a logical device screen belonging to the MIDlet on which the user interface objects or
components can be drawn. A Display can have one or more of these objects to be drawn on the device screen. The base class of these objects or components is represented by the class Displayable. Derived from this base class are the actual displayable objects which are divided into the high-level and the low-level APIs. The super classes for the high-level and the low-level APIs are Screen and Canvas respectively.

The high-level APIs are designed to maintain application portability among similar devices. An application written and tested on a particular device using these APIs will probably run and look the same on another similar device. Consequently, these APIs provide the application programmers with almost no control over the look and appearance of the application. However, on the other hand, the application designers do not have to worry about adapting the application to other similar devices, as the underlying implementation does the necessary adaption to the device hardware and native user interface style.

As said earlier, the super class for the high-level API is the Screen class. Derived from this class are the individual component classes like Alert, List, TextBox and Form. A Form can contain one or more Item driven class objects, which are DateField, StringItem, TextField and many more.

Unlike the high-level API, which restricts the user from controlling the look and appearance of the application, the low-level API is designed for those who wish to make the most out of the available device screen. The application programmers are free to place and control the graphic elements anywhere on the device screen. However, they cannot use the high-level API components while using the low-level API and there is no guarantee that the application will look the same on the other device. The javax.microedition.lcdui.Canvas and javax.microedition.lcdui.Graphics classes implement the low-level API.

The Canvas class is the building block of the low-level API and it provides direct access to the screen of the MIDP device. Since this is an abstract class, in order to use it, we have to derive our class from it and provide an implementation for the virtual functions in it. The most important function of those is the paint method, which actually paints objects, shapes and characters on the device screen.

To allow user interaction in the application, there exists a class called Command. This class makes it possible to add commands to the application interface which the user can use to give the application appropriate instructions. For example, while running an application on a mobile phone, the "Exit" soft-button that the user may use to exit the application is a command. The Command class has appropriate methods for setting a label, the command type and the priority. These commands can be added to any Displayable object and it will automatically appear above the soft-buttons of the phone depending on the priorities set.

To listen to these commands, the MIDP platform provides an interface called the CommandListener. The CommandListener needs to be registered with the Displayable object using the setCommandListener method. Once this is done, whenever a user issues a command to the application, the commandAction method of the CommandListener is automatically invoked and performs the action as programmed by the application designer.

There exists many other classes and interfaces in the MIDP platform which are capable of performing various tasks. However, we are not going discuss them all in detail as the objective here is to give the reader an overview of the MIDP APIs.
**MIDlet and its Life-Cycle**

As said earlier, a MIDP application is called a MIDlet. There is at least one class in a MIDlet that is derived from a MIDP defined abstract base class `javax.microedition.midlet.MIDlet`. This is the main class of the MIDlet from where the MIDlet starts and ends its execution. In other words, the MIDlet life cycle is controlled by this class.

A MIDlet can be in one of four states at a time: loaded, active, paused and destroyed. Figure 2.7 below shows the transitions of a MIDlet between these states.

The MIDP abstract base class `javax.microedition.midlet.MIDlet` has some method that need to be implemented in the derived class and which control the life cycle of the MIDlet. As you can see in the figure, every state transition takes place as a result of a calling these methods. These methods are `startApp()`, `pauseApp()` and `destroyApp()`.

When the MIDlet is deployed on the devices, it is said to be loaded. Just after the device user starts the application, the `startApp()` method is called and the MIDlet goes into the active state. From the active state, the `pauseApp()` function can be called which will pause the application. Alternatively, the `destroyApp()` method can be executed which destroys the application.

To get a feeling of a MIDP GUI application, see [41]. The sample MIDP code shown in [41] reflects all the basics explained about MIDP applications in this chapter until now.

**Symbian Application Development**

In this section we will briefly talk about the Symbian Operating System and will delve into necessary details of application development on Symbian OS. The discussion of later will include an overview of the general Symbian application framework, brief introduction to the two Symbian based UI design platforms (S60 and UIQ), basics of Symbian application architecture and mentioning things that are different in Symbian C++ and general C++. Wherever needed, we will discuss things in relation to the S60 platform, because the S60 platform, as we will see later, was used to develop the DoD Symbian Client. Also, this section will not illustrate and explain sample codes as the intention here is to give an overall understanding of the Symbian platform and it is assumed that the user is familiar with basic programming practices.

**Symbian OS**

Symbian OS is an operating system for battery-powered devices having 32-bit CPUs with slower speeds relative to desktops and workstations, like mobile phones and PDAs. Current Symbian OS systems are based on XScale, ARM9 and ARM11 based CPUs. It is an open operating system.
allowing third-party developers to write and install applications independently from device manufacturers. The native language of Symbian OS is C++ and therefore the native Symbian APIs are C++ APIs. The native Symbian APIs allows the developers to access various phone features such as telephony and messaging in addition to the basic OS functionalities.

The entire Symbian OS can be seen as having a layered system model. Starting from the bottom, each layer uses the functionality provided by the layer beneath it and provides services to the layer above. The figure 2.8 below shows how the Symbian OS layers are stacked.

The UI Framework layer provides the ability to build a variety of user interfaces on top of Symbian OS. This layer encapsulates several components which are responsible for the look-and-feel of the user interface and user interaction. These components are used to develop variants of user interfaces which are further used by the application developers to build applications based on those UI designs. The most used UI design platforms today are S60 (from Nokia) and UIQ (from UIQ Technologies). We will talk more about these UI design platforms later in this chapter.

The Application Engine Layer provides user interface independent services to the applications. These services include providing support for device management, personal information management (PIM), office applications and many more.

The System Layer puts together all the APIs and libraries that provide low-level support to the applications. This low-level support includes support for connectivity, communication, graphics and multimedia.

The Kernel Layer embraces software that is responsible for kernel services, like process scheduling and threading.

The Hardware Adaption Layer is software which performs that task of adapting the upper software layers to the hardware. This layer is dependent on the hardware, but the upper software layers are independent of the hardware and the hardware adaptation layer.

Finally the Hardware Layer represents the actual hardware of the device.
SYMBIAN APPLICATION FRAMEWORK

The most integral subsystem of the Symbian Application Framework is UIKON which facilitates support for GUI applications. It is part of the UI Framework layer and its flexible architecture allows various design platforms to run on top of the core operating system.

UIKON itself is derived from two important frameworks:

- CONE, the control environment, is responsible for graphical interaction
- APPARC, the application architecture, is responsible for applications and their data

Figure 2.9 below shows the basic classes of CONE and APPARC and how UIKON classes are derived from it.

The complete framework shown above depicts the use of Model-View-Controller (MVC) design pattern. The phenomena behind MVC pattern is that the application should be split into several logical parts, each catering to a different aspect of the whole application. The Symbian Application Framework implements this phenomenon by providing separate classes for Model (a document, CEikDocument), the View (an application view derived from CCoeControl) and the Controller (an application UI, CEikAppUi). A UI application must implement these three classes along with an application class (a class derived from CEikApplication) which is the entry point of the application. The document class represents the data model for the application, whereas the purpose of the application view is to represent the data on the screen. The application UI class is responsible for accommodating the application views and presenting them on the screen as well as handling commands that are generated by the user while using the application. In addition to all these classes, an application developer may have to define some element for the application, like the menus, short cut keys, string resources etc. All these elements are stored in a resource file.

Finally, a developer can implement engine classes which drive the application logic and use them in their code appropriately.

UI DESIGN PLATFORMS

On top of the UIKON framework, there exist several UI design/application platforms. These UI platforms extend the existing framework by providing platform-specific controls. Each platform defines a layout of the user interface of the mobile device and the kind of user interaction mechanisms it can accept. So, the developers have to decide beforehand the UI platform on which they will base their application on, as an application designed for one UI platform cannot run on the other.
The two most widely deployed UI platform are the S60 and the UIQ. The UIQ-specific library is called Qikon and the S60-specific library is called Avkon. These platform specific libraries derive their classes from the MVC classes of UIKON. Similarly, any Symbian application, depending on which platform it is being designed for, must derive their MVC classes from the MVC classes of the platform. The figure below shows class derivation of the S60 UI platform and an S60 Symbian application.

![Figure 2.10 Class Derivation of S60 platform and S60 Symbian application](image)

**VIEW ARCHITECTURE**

A view is generally defined as a collection of interface elements, like menus, commands, buttons and controls. All these things collectively describe a particular aspect of an application. When an application is performing a single task, there will be just one view which will have and display all its interface elements required to perform that task. Say, for example, the sole purpose of an application is to store daily expenses of a person, then the view in that application will have some textboxes to collect the data, a save command button to save the data and an exit button to exit the application. However, if the application is also meant to display that data, then this has to be done in a separate view. This view may have some labels and other display elements to display the data.

When there is just one view in the application, the view class in that application can be derived from CCoeControl. However, when there has to be multiple views in an application, view architecture must be used. In that case, the view class is derived from C AkronView and the application UI class is to be derived from C AkronViewAppUi. Figure 2.11 below shows a class diagram for achieving this in S60.
The class diagram above shows that the AppUi class can have one or more views. Also, View class can contain one or more interface elements, which are derived from CCoecControl or from a class which itself is derived from CCoecControl. The AppUi class inherits methods for handling user commands, adding views to the view server and activating and deactivating views. These methods can be used to perform switching of views. Consequently, when a view is activated or deactivated, the activation and deactivation callback methods in the view class are called by the framework to perform appropriate actions.

**DOD Solution**

In this section we will look at the work done in the past on the server and of the DoD project. We will first briefly explain the technological components of the DoD architecture, followed by defining the architecture of DoD and finally the basic operation of DoD will be mentioned. We will not mention about the existing Java client for DoD, as we will talk about it in more detail in the next chapter. We will not delve into the details of the server implementation of DoD as this thesis work focuses on the client side of it. The sole intention of this se

**Components of DoD Architecture**

**GSM Gateway**

GSM gateway is a device that makes use of a GSM SIM card to allow people to make cheap calls to mobile networks from their fixed handsets. When a fixed handset is connected to a GSM gateway, it is possible to make and receive calls from the fixed handle through the mobile network. This eventually reduces the high cost of making calls from fixed phones to mobile phones.

**Asterisks PBX**

Asterisks is an open source software implementation of a telephone PBX. It allows phones connected to it to makes calls between each other and to other networks like public switched telephone network (PSTN) and voice over internet protocol (VoIP).
When a GSM gateway is connected with a PBX system (in case of DoD, it was Asterisks), all calls into the cellular network are routed via the gateway instead of via the PSTN.

**DoD Architecture Description**

The figure 2.12 below shows the architecture and illustrates the basic operation of the DoD Solution.

![Figure 2.12 Architecture and Operation of DoD Solution](image)

The main objective of saving call costs is achieved by the inclusion of the GSM gateway. GSM gateways are used when the objective is to save cost for calls originating from fixed line to GSM networks [42]. It is equipped with GSM radio antenna and multiple SIM cards which are used by intelligent software. In practice, GSM gateways are used in conjunction with company PBXs to convert all the outgoing fixed-to-mobile calls to mobile-to-mobile calls. For the DoD solution, GSM gateway is combined with the DoD server and Asterisk PBX, which introduces the mobility required for the solution. The DoD server accepts call requests from mobile and web clients over the internet and sends the received information to Asterisk PBX for placing appropriate calls. The Asterisk PBX initiates two call requests to the GSM Gateway which are later bridged. The GSM Gateway, if proper configurations are done, place calls according to Least Cost Routing (LCR), which means redirecting calls to the same network to achieve the minimum cost for the phone calls.

**DoD Solution Operation**

As shown in the figure above, a call is initiated in step 1 by the caller and this is done using a mobile interface (which is called the DoD Client). The call initiation is basically a request sent to the DoD server over the data network, i.e. GPRS. This call request contains the number of the caller, the number of the callee, callers DoD account username and the password. This request is a simple HTTP post request having a particular format, which is only understood by the DoD server. This is illustrated by steps 2 and 3. When the request reaches the DoD Server, and as
shown by the steps 4 and 5, the DoD server authenticates the callers username and password in the database. Once the caller is authenticated, the Asterisks PBX places a call via the GSM gateway to the caller (steps 6 and 7). After the call to the caller is connected, another call is originated from Asterisks PBX to the callee, again through the GSM gateway in steps 8 and 9. When the callee accepts the call, the last step is for Asterisks to bridge the call.

The purpose of the DoD server is not only to authenticate the caller, but it also performs a Least Cost routing algorithm to ensure that cheapest way of making the call is selected. Outgoing calls can not only be made to mobile networks, but also to PSTN networks and to VoIP phones.
3) **EXISTING DoD CLIENT**

In this chapter we will present an analysis of the existing DoD client application which was implemented using J2ME by Max Weltz [39]. This analysis will be done on the basis of application design, some important classes and the application usage. The basic purpose of this analysis is to analyze the shortcomings of the existing client application. These shortcomings will then become our main requirements for the new DoD client.

**ANALYSIS: J2ME DoD CLIENT**

**CLASS DIAGRAM**

Below is the class diagram of the DoD Client application that was implemented in J2ME by Max Weltz [39].

![Class Diagram](image)

The class diagram above conforms to the explanation of MIDP GUI applications and the class structure shown in figure 2.6 in Chapter 2. Since it is a GUI application, it contains a class derived from MIDlet, which is DoDClient. As shown in figure 2.6, a MIDlet can have zero or more displayable objects, i.e. classes derived from Displayable. Accordingly, the class DoDClient is associated with the class CallCanvas, which is derived from Canvas, which is derived from Displayable. HTTPPostConnection class handles communication with the DoD server and is associated with CallCanvas and DoDClient. The class Prompt deals with prompts on the application screen and are associated with CallCanvas. Lastly, the class PreferenceStore deals with the storage of user data used for making call back calls, like call back number, server IP address, server port etc.
IMPORTANT CLASSES

**DoDClient**

As mentioned in chapter 2, under J2ME Application Development, each J2ME GUI application is called a MIDlet and must contain at least one class derived from MIDlet. The class DoDClient is derived from the MIDlet class and the CommandListener interface and is therefore the most important class in the application. Since it is derived from MIDlet, it is the start and end point of the application.

The class MIDlet has 3 methods which are inherited by DoDClient. These methods are `startApp()`, `pauseApp()` and `destroyApp()` (see chapter 2). These methods are called by the framework to start, pause and destroy the application respectively. When the user starts the application, the `startApp()` method is called by the framework. If, while the application is running, it is interrupted by any other application or event, for example by an incoming call, then the `pauseApp()` method is called to pause application. Lastly, when the application is closed by the user, the `destroyApp()` method is called to destroy the application.

From CommandListener interface, the DoDClient inherits the `commandAction(Command aCommand, Displayable aDisplayable)` method which is called by the framework whenever the user selects an option from the menu. Menu options are commands and commands can be added to any item whose base class is Displayable. The `addCommand` method of class Displayable is used to add commands. The parameter `aCommand` of method `commandAction` tells which menu option is selected by the user. The argument `aDisplayable` tells the displayable items to which the command is related to.

Besides the inherited methods, DoDClient contains some methods of its own. The most important of all is `submit()` which submits a post to the DoD server using an instance of class `HttpPostConnection`. Remaining methods of DoDClient deal with displaying appropriate forms on the screen. For example, the `showSetup()` method displays a settings page on the application screen. It adds five text fields and some commands to a form and sets it to display. Other similar methods are `showPhoneBookForm()`, `showConferenceForm()`, `showCallForm()` etc.

**CallCanvas**

The class CallCanvas is derived from Canvas and is used in the application to display a call form. An instance of CallCanvas is therefore used in the method `showCallForm()` of class DoDClient. Inside this class, there are methods which mainly deal with drawing on the application screen. For example, the `paintConference(Image screen)` method creates an image with conference call GUI layout, the method `setMessage(String m)` paints a message on the screen and so on.

The methods `paint(Graphics g)` and `keyPressed(int keyCode)` are inherited from Canvas. The method `paint(Graphics g)` is called by the framework whenever the canvas is set on display using the `setCurrent()` method of Display class. It is called when the application code requests a repaint of the application screen using the method `repaint()`. The method `keyPressed(int keyCode)` is also called by the framework whenever the keys on the phone are pressed. The integer argument `keyCode` tells which key was pressed.

**HttpPostConnection**

The HttpPostConnection class is a thread class and as its name suggests, it deals with sending HTTP post requests to the DoD Server. Since, it is a thread class, it extends the Thread class.
From Thread, it inherits the method run(). This method makes a call to the only other method in the class, which is submitAnswer(). In this method, the application sends the call back request to the DoD server and reads its response and displays it on the screen.

APPLICATION USAGE

Once the application was installed on phone, it had to be opened from applications folder in the phone where it was installed. The figure below show what the Java client application looks like.

![Figure 3.2 Java DoD Client Application GUI](image)

After it is opened, the user can use the phone keypad to dial in the number on the application screen and the number will appear in the white box show in the figure. After typing in the number, the user can use the "Call" option in the application screen to dial the number. The applications settings option and other options can be accessed using the "Menu" button.

The major drawback here is that this is the most un-natural way of making phone calls. Every time the user wants to make a call using the DoD solution, he/she has to browse into the phones application folder, locate the application icon, open the application and then make a call. Once the application is open, it restricts the user from using any other phone feature. In order to use the phone for some other purpose, e.g. using internet, playing games etc, the user has to close the DoD application.

SHORTCOMINGS AND REQUIREMENTS FOR DO D CLIENT

In this section we are going to deduce the shortcomings of the existing DoD client from the analysis that we did on it in the last sections. Based on the shortcomings, we will introduce the areas which need to be improved in the new DoD client.

USER FRIENDLINESS

The main goal that OptiCall wanted to achieve from the DoD project was twofold; reducing their customers call cost by a fraction of up to 90% along with an easy to use application on their phones. The first part of their goal was achieved by the DoD server as shown in the DoD architecture in the previous chapter. However, the second element was not completely fulfilled by the J2ME DoD client. As we just saw in the last section, the J2ME DoD client lacked user friendliness. Every time the users wanted to make a call from the DoD client, they will have
browse into the folder where the application was installed, open the application i.e. let it start, load any graphics etc, then dial the number on the client application screen, press one of the menu buttons and then press the Call option. Even though the Call operation was performed quickly, all the work that a user had to do before making a call, was a very tedious job. Also, once the application was running, the user cannot do anything else with his phone, e.g. play a game, view the calendar etc. If the user has to do anything else than making a phone call, the DoD client application has to be shut down. And when the user wants to make a call again, the whole tedious job has to be done again.

Incorporating user friendliness, therefore, becomes the major requirement for the new DoD client application. The new DoD client application should, ideally, not change the normal user behavior while making phone calls. It should only become active when the user wants to make a call. We are therefore going to work towards making an application which is transparent to the user but is functional at the same time.

APPLICATION DESIGN

Apart from the major requirement of user friendliness, there were some other areas of improvement which were to be taken care of while developing the new DoD client. Most important of them was the object oriented design of the application. Having analyzed the J2ME DoD client in the previous sections of this chapter, it is evident that the application design is not well structured. All the functionalities of the application have been programmed as functions into only a couple of classes which are not even named properly. For example, the class DoDClient implements the Conference functionality by implementing the function showConferenceForm (). So, instead of implementing the Conference functionality as a class in the application, it has been implemented as a function of the DoDClient class. This also leads us to believe that the application is not easily extensible. For this, a lot of concentration has to be put into the application design while developing the new application.

APPLICATION PERFORMANCE

The other utmost important things to improve are the measures that are directly linked with the application performance. These include the application start time, application security, affects on the battery life of the phone etc. So, while developing the new DoD client we must make sure that in the quest of achieving the major requirements of the application, i.e. that user friendliness and the application design, we should not only compromise on the application performance, but also strive to improve it.
4) **Capability Analysis: J2ME vs. Symbian C++**

In this chapter, we are going to compare, in the light of the requirements presented in the last chapter, the capabilities of the J2ME and the Symbian platforms. The comparison will be done in terms of some performance indicators and the inherent APIs present in these platforms which can fulfill the needs of the DoD client application. Based on this comparison, we will decide which of the two platforms should be chosen for developing the new DoD client.

**Comparing Application-Specific Performance Indicators**

The main source of information considered for comparing application-specific aspects of J2ME and Symbian was an article published by Symbian, titled *Native and Java ME Development on Symbian OS* [17]. This article limited the advantages and disadvantages of using J2ME and Symbian C++ for application development on Symbian OS only and not for other feature phones. However, as we will see soon, the facts presented in this article for J2ME are such that will hold true for feature phones also.

**Runtime Speed of Application**

Since C++ is the native language of Symbian OS, the Symbian C++ code is compiled directly into ARM instructions. Therefore, there is no need for an intermediate interpreter at the runtime. On the contrary, as we said earlier, it needs a Java Virtual Machine (JVM) to be installed on the system to run Java applications. The JVM, or KVM in case of J2ME applications, performs a runtime interpretation of Java instruction codes to native language before the proper set of ARM instructions is executed. This clearly gives Symbian C++ an advantage over J2ME in terms of runtime speed of the application.

**Application Start Time**

Presence of an intermediate interpreter also adds to degrade application start time. Every time a Java application is started, its Virtual Machine needs to be loaded as part of application initiation. Combining the start time of both, the JVM and the application itself, it takes about 4-6 seconds to have a J2ME MIDlet running. Also, if the application is complex and requires a large amount of data or heavy graphics to be loaded, then this figure increases further. A Symbian application written in the native language does not suffer from the extra time of loading an intermediate application/interpreter. Hence the application start time for Symbian application is very much quicker than a J2ME application.

**System APIs**

Symbian C++ being native to Symbian OS has full access to low-level system APIs and to the low-level functionalities of the Symbian OS and the phone. J2ME, on the other hand, being non-native and an external development platform, does not have access to all the functionalities a platform has to offer, including the Symbian OS. Java (and J2ME) APIs are present in the form of JSRs (Java Specification Requests), which are published by JCP (Java Community Press). Apart from the fact that JSRs do not provide the same access to system functionalities as the Symbian C++ APIs, JSRs are usually linked to phone models. A particular JSR may not be supported on a phone for which an application is planned. Having said this, usage of low-level APIs can bind a developer of to a specific platform, which may create portability issues.
However, it mainly depends on the intended functionalities of the application and its market when deciding on a particular platform on the basis of system API access. In our case, as per OptiCalls requirement of a transparent/background application and as we will see further on, it was evident that we will need complete access to low-level system APIs to achieve the goals to the DoD client.

**APPLICATION SECURITY**

The security model of J2ME devices is such that it can affect the usability of Java applications at runtime and during installation. Security policies to control access to protected APIs often overload the application with user prompts. In majority of the cases, these user prompts ask the user to grant/deny application access to phone functionalities like message, application auto start, file read/write, camera and network functionalities. Additionally, extra layers of security can added from the manufacturer and/or the operators' side. This further complicates the applications' usability.

Symbian, on the other hand, has a security program for native C++ applications called Symbian Signed. Every native Symbian application goes through the Symbian Signed's application signing process before it is deployed on phones for commercial or test purposes. While the application is being developed, the application developer adds Symbian defined capabilities to the application. These can be ReadDeviceData, NetworkServices, ReadUserData, WriteDeviceData etc. Selection of specific capabilities, say NetworkServices, implies that the application developer has made use of network services, for example use of internet. When the application is sent for signing to Symbian Signed, it is tested based on those capabilities and then the application is signed. Once signed, when the application is being deployed on the phone, all the checks are performed at the time of installation and none at the runtime. This, along with being signed by Symbian as a secure-to-use application, enhances the usability of the application and guarantees pleasant user experience.

**APPLICATION UPDATE INSTALLATION**

The development of most computer and mobile applications does not end once the application is commercialized. Their development continues, even after the application is in the market, to introduce new features to them and improve their existing features. This results in various versions of the same application in the market. The most efficient way of installing the latest version of the application is to have an update package, which does not require overwriting or deleting the already existing version but update the existing version, keeping the data in the older version intact with the newer version. J2ME applications lack this ability. The J2ME applications’ new version has to be overwritten the older version. On the contrary, native Symbian C++ applications can be updated to newer versions via the package type of the SIS file (SIS file is a Symbian C++ application installation file with .sis extension). This ability of updating the application to a newer version allows a variety of after-sales support options when releasing patches and updated for the application.

**BATTERY AND MEMORY USAGE**

Since Symbian OS is specifically designed to cater the needs of mobile devices with limited resources and very little memory, various Symbian OS programming idioms and features emphasize on conserving memory. Symbian OS provides two concepts called Leaves and Cleanup Stack. Leaves provide a mechanism for exception handling, which uses fewer resources than normal C++ or Java exception-handling. Cleanup stack is a static resource which is used to clean up dynamically allocated resources on the system heap in case of leaves. These two concepts provide fine grain control over memory and resource usage. Even though J2ME is a
specially condensed version of Java Standard Edition for mobile phones, most of its programming practices are similar to that for desktop applications and which are catastrophic for mobile devices. Also, the memory management practices of languages also affect the battery usage of phone. Consequently, the better the memory management, the better would be the battery usage.

**COMPARING CRITICAL APIs**

In this section we will compare the J2ME and the Symbian APIs which are most likely to be used to achieve the needs of the DoD client.

Since DoD is a telephony application and the most important requirement deduced for DoD client is to not change the default behavior of the phone while making calls through DoD server, the most obvious APIs to analyze for a possible solution to the problem were those which dealt with Telephony. Below we discuss the capabilities provided by the telephony APIs of J2ME and Symbian.

As said in chapter 2 while describing the DoD solution, an HTTP request needs to be send over the data network to the DoD server. From this we know that the application would have to use some API which handles the HTTP transactions to send the request. Hence, HTTP programming APIs also needed to be compared from Symbian and J2ME.

Lastly, based on the requirement of a transparent application, we needed to look for an API, if any existed, in J2ME and Symbian would let the application run in the background and still be functional.

It is important to note here that this section will not describe how the APIs will be used, but will only compare the J2ME and Symbian APIs in terms of their capabilities.

**COMPARING HTTP PROGRAMMING APIs**

As mentioned in chapter 2, the request that is sent to the DoD server is a simple HTTP POST request which is only understood by the DoD server. J2ME and Symbian both have APIs that collectively deal with all aspects of HTTP transactions, starting from creation of HTTP connections to sending of HTTP requests to closing of connections. Even though the APIs in J2ME and Symbian eventually perform the same operation, there complexity and control over the operations varies. We will first look at HTTP specific APIs in J2ME and then that of Symbian.

In J2ME, the collection of interfaces that handle the low-level network connections is called the Generic Connection Framework. The figure below shows the hierarchy of GCF:
As it is obvious from the figure, the GCF contains an interface that handles the HTTP connections, that is the HTTPConnection. Besides HTTPConnection, there are 3 other APIs/classes that are involved in HTTP transaction: the Connector, the InputStream and the OutputStream.

As mentioned in an article [18] published by Sun Microsystems, the **Connector** class is a factory of connections. Calling the Connector.open() method would return an object of type Connection.

An example of calling this method that should return an **HTTPConnection** is:

```
```

Once the above function call is executed, we have established a connection, in our case, to www.yahoo.com. Now, we are ready to send request to and read responses from this URL. The mechanism for sending request to and reading responses from the URL comes further below.

Some important functions in the HTTPConnection are listed below:

- **getRequestMethod()** - Gets the HTTP request method, e.g. GET, POST, HEAD, etc
- **setRequestMethod(String method)** - Sets the HTTP request method, e.g. GET, POST, HEAD, etc
- **getResponseCode()** - Gets the response code of the HTTP response message, e.g. 200 OK
- **getResponseMessage()** - Gets the HTTP response message
- **getURL()** - Get the URL to which the HTTP connection currently exists
- **getRequestProperty(String key)** - Sets the request property
- **setRequestProperty(String key, String value)** - Gets the request property

Besides these functions, there are several functions that are imported from the base classes. Among these, there are 2 important functions imported from interfaces InputConnection and OutputConnection. These 2 functions are:
These functions fulfil the need of sending requests to and reading responses from the connected URL. As we can see from the function definitions, these functions return instances of InputStream and OutputStream respectively. An example usage of these functions is shown below:

```java
InputStream is = httpConn.openInputStream();
OutputStream os = httpConn.openOutputStream();
```

Calling these functions in the above manner, the InputStream and OutputStream linked with httpConn is stored in `is` and `os` respectively. We can call the `read()` and `write()` functions of InputStream `is` and OutputStream `os` respectively, to read responses from and send requests to the connected URL.

This is how the HTTP transactions are dealt with in J2ME. Now we'll take a look at how HTTP transactions are handled in Symbian C++.

The HTTP framework in Symbian OS contains several APIs which collectively handle the entire life cycle of HTTP transactions. In Symbian OS, an HTTP transaction starts by opening a session using the RHTTPTSession API. This is done by calling the OpenL() function of the RHTTPTSession API. After the session is opened, the session properties like the connection info and the session headers are set using other appropriate function of the same API. The figure below shows how this is done:

```c
// Open RHTTPTSession with default protocol ('HTTP/TCP')
iSession.OpenL();

// initialize handles
User::LeaveIfError(iSocketServ.Connect());
User::LeaveIfError(iConnection.Open(iSocketServ));

// set them for use with open http session represented by iSession
RStringPool strP = iSession.StringPool();
RHTTPConnectionInfo connInfo = iSession.ConnectionInfo();
connInfo.SetPropertyL(strP.StringF(HTTP::HttpSocketServ,
   RHTTPSession::GetTable()), HTTP::HdrVal(iSocketServ.handle()));
TInt connPtr = reinterpret_cast<TInt>&(iConnection);
connInfo.SetPropertyL(strP.StringF(HTTP::HttpSocketConnection,
   RHTTPSession::GetTable()), HTTP::HdrVal(connPtr));

// Setup the usual headers and indicate that the body will be multipart
RHTTPTHeaders httpHeader = iSession().RequestSessionHeadersL();
SetHeaderL(httpHeader, HTTP::EUserAgent, EUserAgent);
SetHeaderL(httpHeader, HTTP::EAccept, EAccept);
SetHeaderL(httpHeader, HTTP::EContentType, EMultpartContentType());
```

Figure 4.2 Creating a HTTP session

After the session is opened and the session properties are set, the HTTP transaction is created and submitted. The creation and submission of HTTP transaction is done using the RHTTPTTransaction API. An example of this is shown in the figure below:
In the above example, an HTTP post transaction is created by calling the RHTTPSession API's OpenTransactionL () function which returns a transaction that has properties same as the arguments of the function. After the transaction is created, the SubmitL () function of the RHTTPTransaction API is called to submit the transaction.

As we see in the example above, just before the transaction is submitted, the SetBody () function of RHTTPRequest API (the Request () function of RHTTPTransaction returns an instance of RHTTPRequest API) is being called. This function is called to supply the body of the transaction. It takes an instance of MHTTPDataSupplier interface. This interface is responsible for holding and supplying data to the HTTP transaction.

Another interface associated with HTTP transactions is MHTTPTransactionCallback. The purpose of this API is to monitor the progress of the transaction. It's MHFRunL () function is asynchronously called by the framework to take appropriate actions on various transaction events.

Almost always we need to derive our HTTP class in our application from the two above mentioned interfaces, MHTTPDataSupplier and MHTTPTransactionCallback and override their functions in our class. By overriding their functions, these interfaces behave in the manner we want them to. For example, by overriding the MHFRunL () function of the MHTTPTransactionCallback interface, we implement the behaviour of our application to different HTTP transaction events.

As you can see from the descriptions above, the HTTP framework of Symbian OS is more structured than that of J2ME and hence it provides more control over the operations. However, essentially both provide the same functionality.

**Comparing Telephony APIs**

As per the requirements of Opticall AB for a DoD mobile client application, it was obvious that we will need to use APIs that provide access to low-level telephony functionalities. So, in this section, we will look at telephony APIs from Java and Symbian platforms and the depth of functionalities provided by them.

Java platform has two APIs that deal with telephony, the Java Telephony API (JTAPI or JSR 043) and the Mobile Telephony API (MTA or JSR 253).
The Java Telephony API and the Mobile Telephony API provide more or less the same functionality; however, both of them are unusable for our new DoD client application. The JTAPI is not supported on any of the MIDP devices and is only usable for desktop applications. The MTA, on the other hand is targeted for mobile devices, but there are no mobile phones in the market to date, which support this API. Hence, it is impossible to get access to the low-level telephony related functionalities of mobile phone using J2ME primarily because there is no such API in J2ME that can do that.

Symbian, on the contrary, has two such telephony APIs which cater to almost all the aspects of calling handling. The APIs are ETel 3rd Party API and the ETel Core API.

The ETel 3rd Party API is built on top of the ETel Core API to restrict access to telephony services provided by the later, and is only a subset of the later [20]. It is available on Symbian S60 3rd edition onwards. The ETel Core API is still available on S60 3rd Edition and the earlier editions. We will briefly go through the capabilities provided by these APIs.

Present in ETel 3rd Party API is a class called CTelephony. This class, as described in [19], provides a simple interface to the phone's telephony subsystem. The only two services provided by this class are 1) finding out information about the phone and 2) dialing, answering and controlling voice calls.

The ETel Core API provides a bit more access to the phone functionality than that ETel 3rd Party API. This API [21] defines an interface that allows clients to access a limited set of almost universally accepted telephony functionality. This API has 4 main classes: the RTelServer, RPhone, RLine and RCall classes.

The RTelServer is an interface to the Root Session which provides access to the system telephony information [21]. The client must open a sub-session to the phone device, before trying to access the functionalities associated with it.

The RPhone class provides an abstract interface to a particular telephony device [21]. The interface provides functions to access the status and capabilities of the device and to notify changes to the same.

A telephony device can have one or more lines [21]. The RLine class provides an interface to the lines. It contains functions to access the status and capabilities of the line and notify changes to the status and capabilities of the line.

A line can have zero or more calls. Interface to the calls is provided by RCall. This class simulates the functionalities of a call, i.e. dial a number, wait for and detect incoming calls, detect changes to the state of a call and hang up a call. Along with this, it provides methods to get status and capabilities information and be notified of the changes.

COMPARING APIs FOR SENDING APPLICATION TO BACKGROUND

In Symbian, there exists a class called TApaTask which can be used achieve the objective of sending an application to background. This is an application task class and its description in [23] is as follows:

A task is a running application. At any one time, for any given application, there may be zero, one, or more tasks running on a device.

A task is identified by its association with the running application's window group.
This class has following important functions:

**TApTask (RWsSession &aWsSession)** – this is the class constructor and it takes an instance of RWsSession; RWsSession represents a window server session and every application has a window server session. We will not discuss the details of RWsSession class here. For more information on this class see ref[24].

**BringToForeground ()** – this brings the task to foreground

**SendToBackground ()** – this send the task to background

**SetWgId (Tint)** – this method sets the task’s window group Id

Using the above functions, it is possible to send an application to background. We are not going to demonstrate how this is done programmatically, as the only purpose here is to show that it is achievable. To read more about the implementation details, see ref [25].

**DECISION**

Based on the requirements presented in the last chapter for the new DoD client and the result of the analysis done between J2ME and Symbian C++ in the last section, it is obvious that the Symbian platform is capable of giving access to many low-level APIs which is highly desirable for the new DoD client. Also, it is possible to have a better performing client application, in terms of security, application start time and other measures of performance as discussed in last section, within the Symbian platform. Therefore, based on these conclusions it was decided to have Symbian C++ as the base technology and platform for the new DoD client.
5) **SOLUTION AND IMPLEMENTATION**

In this chapter we will look at the solution devised for OptiCalls’ requirements based on the study carried in the last chapter. We will then go through the technical details of the implementation of the solution. The technical details that we will present here will only cover the high level details of the application architecture, the development environment and description of some important classes. We will refrain from discussing the entire application code. The application code will be provided with this thesis report. Lastly, we will describe the main features of the application and show how the application will work.

**SOLUTION**

The thorough analysis of the Symbian APIs in the last chapter helped to devise a solution for the requirements of the DoD client. Out of all the conclusions driven from that analysis, the following four can be used to fulfill the requirements of DoD client:

1) Symbian applications can be sent to background and brought back to the foreground.
2) Changes in call state can be monitored using the ETel Core API.
3) It is possible to hang up existing and dial new calls from an application using the RCall class.
4) It is possible to send HTTP requests from Symbian applications using the HTTP API in Symbian, i.e. the RHTTPSession, RHTTPTransaction and RHTTPRequest classes.

We can use the above four facts to create some C++ classes and methods to devise the solution. So, fact 4 can be used to create a class that will handle all the communication with the server that is setting up connection to the DoD server, sending http requests to it to place call back calls, reading responses from the server and responding to them.

Another class can be created using facts 2 and 3. This class can contain methods to observe outgoing calls and opening and hanging up those calls, and then also sending the call back request to the DoD server using the class mentioned above. It can also contain other functions related to the mandatory resource allocation and de-allocation which might be needed while using the ETel Core API (as it is one of the low level APIs in Symbian).

A method or a class can also be implemented which handles the sending of the application to the background and bringing it back to the foreground.

Having implemented these classes, we can give the user an option in the application menu to send it to the background. As soon as this option is selected, the outgoing call listener can be started and the application can be sent to the background. The class CAknViewAppUi, as discussed in chapter 2, has an event handling function called HandleForegroundEventL () [26] which is automatically called when the application focus switched to/from the foreground. So, when the user will dial a number, the call can be hanged up by the outgoing call listener and the application can be brought to the foreground. Consequently, the HandleForegroundEventL () function will be automatically called, and there the call back request to the DoD server can be sent. Once the request is sent, the application will be again sent back to the background. In one of the sections below we will see how this application can be used from the users view point.

The sequence diagram below shows how the different entities in the application will interact.
IMPLEMENTATION

DEVELOPMENT ENVIRONMENT

Development of the DoD Client was done in the Windows environment. Installed on it was Carbide C/C++ IDE and SDKs for Symbian S60 3rd Edition, 3rd Edition FP 1 and 3rd Edition FP2.

Mobile phones used for testing the application was Nokia E61 and Nokia E51.

CALL THROUGH - NEW CALLING FEATURE

During the implementation of the solution, Opticall made an addition to the requirements of the DoD Client. This addition was the implementation of a new calling method termed by them as Call Through. Using this method, the user will be able to dial into a PBX extension directly,
without having to manually dial into the PBX first and then dialling the extension. How this feature is implemented, set up and used in the application is described in the next sections.

**CLASS DIAGRAM**

All GUI based Symbian applications having more than one view, are supposed to use the view architecture. The application, therefore, has to have five mandatory classes derived from those shown in figure 2.11, i.e. CAknApplication, CAknDocument, CAknViewAppUi, CAknView and CCoeControl. If an IDE is used which has an Application Wizard, then these classes will be generated by the wizard. A Symbian application that has a minimum of these five classes will have a single view. To add more view to the application, more CAknView and CCoeControl derived classes can be implemented. To read more about the view architecture, refer to [15], [27] and [28].

The diagram below shows the class diagram of the DoD Client Application.
Figure 5.2 Class diagram for Symbian DoD Client Application
The class diagram of DoD Client Application above conforms to the basic structure of a Symbian application using view architecture shown in figure 2.11. It has the five main classes, that is, CDoDClientApplication (derived from CAknApplication), CDoDClientDocument (derived from CAknDocument), CDoDClientAppUi (derived from CAknViewAppUi), CDoDClientAppView (derived from CAknView) and CDoDClientAppContainer (derived from CCoeControl). As mentioned earlier also, together these classes create an application with a single view, which is, in our case, the main (or welcome) screen of the application. Since the application have more than a single view, the application has more CAknView and CCoeControl driven classes. For example, the CDoDHelpView class is derived from CAknView and it contains CDoDHelp class which is derived from CCoeControl. The same holds true for CDoDContactsView and CDoDContacts, CDoDWhiteListView and CDoDWhiteList and CDoDHelpMenuView and CDoDHelpMenu.

The CDoDSettingsView class has an exception. It is not associated with a single container class that is derived from CCoeControl. However, it is associated with four container classes derived from CAknSettingItemList. (Add in footer with *) (A CAknSettingItemList [29] is a list of different types of CAknSettingItems’ [30], like CAknTextSettingItem [31], CAknIpFieldSettingItem [32], CAknPasswordSettingItem [33] etc.) But, as it can be seen in the diagram, the CAknSettingItemList is derived from another class that derives from CCoeControl. So, CAknSettingItemList is therefore derived from CCoeControl and consequently the four container classes, that is CDoDSettings, CDoDCallBackSettings, CDoDCallThroughSettings and CDoDOperationalSettings, are derived from CCoeControl. These four container classes, as their names suggest, are used to display particular settings in the application. Switching between the containers is performed by the CDoDSettingsView class depending on the menu option selected by the user. More about the application usage will follow in the section later.

All the classes discussed above deal with the look and feel of the application. Besides these, there are some classes that deal with the data in the application. For example, the CDoDSettingsData stores DoD server settings data like IP Address, port number etc, and the CDoDCallBackSettingsData will store the call back numbers and so on.

Lastly, and most important of all are the classes that implement the most basic and important requirements for the application, and can be called the engine classes for the application. These are the CDoDServerCommunication and the CDoDTransparencyEngine classes. The CDoDServerCommunication class incorporates methods that help communication with the DoD Server and is derived from MHTTPTransactionCallBack and MHTTPDataSupplier interfaces [34]. The CDoDTransparencyEngine class implements the idea of transparent application as mentioned in the last section. This class is derived from CActive [35]

DESCRIPTION OF IMPORTANT CLASSES AND THEIR METHODS

**CDoDClientAppUi**

This class in DoD Client application handles all the GUI related tasks, which includes keeping track of all the views in the application, switching between the views and responding to user actions. As can be seen in the diagram above, this class derives from CAknViewAppUi and so it inherits some important methods from the base class which need an implementation in CDoDClientAppUi. Besides these methods, the class has its own methods and member variables. HandleCommandL (TInt aCommand) and HandleForegroundEventL (TBool aForeground) are two most important derived methods, and SendToBackground () and BringToForeground () are the other two important self declared methods.
HandleCommandL method handles all the user commands. That is, whenever the user selects an option from the application menu, this method automatically gets called by the framework. The argument aCommand tells which command was selected by the user. The switch statement in the method implementation provides a case for each possible value of aCommand. So, say for example, the user selected server settings from the application menu, then the case that corresponds to server settings is executed and the proper view is activated.

The HandleForegroundEventL (TBool aForeground) gets called automatically by the framework whenever the application goes to background or comes back to foreground. The argument aForeground tells whether the application just went to background (aForeground being false) or it just came back to foreground (aForeground being true). Interesting to us is the case when the application comes back to foreground after being sent to background. This can happen in one of the two ways. One is when the user dials a call which gets intercepted by the application and the application is brought back to foreground. If that happens, then the Operational Settings are interpreted by the application and the call is handled appropriately. For example, if the Operation Settings indicate that a call back call should be made, then a request is sent to the DoD Server containing the dialed number and the caller's number, and the application is again sent to the background. Other possible Operational Settings can be Call Through call, Standard Call or Always Ask. Operational Settings are discussed in more detail in the section below.

The SendToBackground () sends the application to background and BringToForeground () brings it back to foreground. It is when these methods are called, the HandleForegroundEventL method is called. These methods are implemented using the technique discussed in Chapter 3.

**CDoDServerCommunication**

Communication with the DoD Server takes place when a call back request needs to be sent to it. The call back request is actually an HTTP POST request with several parameters that the server reads and sets up the call. In order to achieve this, the CDoDServerCommunication class implements an HTTP wrapper class and provides methods to connect to and communicate with the DoD server. This class is derived from MHTTPTransactionCallBack and MHTTPDataSupplier and therefore inherits and provides implementation for their methods.

The methods SetupConnection (), SubmitPostToServer (const TDesC8& aUri, const TDesC8& aBody) and CloseConnection () together allow the application to connect to the DoD server, create a HTTP transaction and submit it to the server and then close the connection. These three the most important methods defined by the class itself.

The class inherits MHFRunL (const RHTTPTransaction& aTransaction, const THTTPEvent& aEvent) and MHFRunError (Tint aError, const RHTTPTransaction& aTransaction, const THTTPEvent& aEvent) methods from the MHTTPTransactionCallBack [36] interface. The MHFRunL method is called by the framework, after the transaction has been created and successfully submitted, in response to the occurrence of a transaction event. The argument aEvent of type THTTPEvent in the methods above specifies the event that has occurred. Once we know which event has occurred, we can provide an implementation for the action to the event that has occurred in a switch statement. The MHFRunError is called when an error occurs in MHFRunL method.

CDoDServerCommunication class also inherits four methods from the MHTTPDataSupplier interface. These four methods deal with data in the HTTP transactions. These functions are
GetNextDataPart (TPtrC8& aDataPart), ReleaseData (), OverallDataSize () and Reset (). Read more about MHTTPDataSupplier and its methods on [37].

**CDoDTransparencyEngine**

This class implements the most important part of the solution devised for transparent application in the “Solution” section, which is to listen to outgoing calls and hanging them up, and therefore it heavily uses the classes RTelServer, RPhone, RLine and RCall provided by ETel Core API (see chapter 3). Since this class should monitor outgoing calls at all times, it is derived from CActive ((write this in footer) CActive derived class objects are called Active objects. Read more about active objects in [11],[38] and [39]).

This class implements several important methods; InitializePhone (), CloseAllResources () and OutgoingCallObserver (). The InitializePhone () allocates phone resources. This is done by first creating a connection to the root session, loading the phone module, and then opening a sub-session to the phone device and then opening the phone line (see the discussion on telephony API in Chapter 3). CloseAllResources () method releases all these resources. The OutgoingCallObserver () method initializes the phone using InitializePhone () method and then calls the NotifyStatusChange (TRequestStatus& aStatus, enum TStatus& aCallStatus) method of RLine class to monitor the status of the phone line.

Besides these methods, the CDoDTransparencyEngine class also inherits two methods from CActive; the RunL () and the DoCancel () methods [38]. The RunL () method is called by the framework when a status change is notified by the NotifyStatusChange (TRequestStatus& aStatus, enum TStatus& aCallStatus) method. So when the argument aCallStatus changes its value, the RunL () method is called, and within that function, an appropriate action is taken depending on the value of aCallStatus. Therefore, when the value of aCallStatus indicates that a call is being dialled, the call can be opened using the OpenExistingCall method of RCall class and the called can be handed up using Hangup() method of RCall class (see Chapter 3). The DoCancel () method cancels the status change notifications by calling NotifyStatusChangeCancel ()

**CDoDCallThrough**

This class implements the new calling method of Call Through using the ETel 3rd Party API class CTelephony (see Chapter 3). The two most important methods in this class are the MakeCall (const TDesC& aNumber) and SendDtmf (TDesC& aSequence, TTTimeIntervalMicroSeconds32 aInterval, TInt aDTMFCount). The MakeCall method uses the DialNewCall method of CTelephony class to dial a new call. It is used when the application dials a call to a PBX number.

The SendDtmf method uses the SendDTMFTones method of CTelephony class to send DTMF tones. It is used by the application to dial in the pin code or the extension number dialled by the user after having dialled into the PBX using the MakeCall method.

So, when a user wants to make a CallThrough call, he/she simply dials the extension number and application first dials into the PBX number using the MakeCall method, then dials the PIN number, if there is any, using the SendDtmf method and then finally dials the extension, again using the SendDtmf method.
This class is also derived from the CActive, so it inherits the RunL() and the DoCancel() methods. However, because this class doesn’t need to perform some task continuously, the RunL() method is empty.

**INSTALLATION, FEATURES AND USABILITY OF DoD CLIENT**

**INSTALLATION**

Installation of the DoD client is done using a .sisx or a .sis file. These files are generated as a result of the DoD client source code compilation and can be found in the sis folder inside the application folder. After the .sisx or .sis file is generated, it can be transferred to the target Symbian phone using Bluetooth, data cable or any other available means. When this file is ran from the saved location, the installation starts automatically and the application will be stored in the mobile phones’ default applications folder, which is normally different for each phone. This installation process is used by the development team of the DoD client for the purpose of development only. For using it commercially, please contact Opticall AB.

**APPLICATION FEATURES**

The two main features of the DoD client application are its two different methods of calling; the Call Back and the Call Through.

If a call is made using the Call Back method, the caller receives a call on his/her mobile phone from the DoD server. After the callee accepts the call back, the DoD server will call the callee. If the callee accepts the call, then both the calls are connected.

If a call is made using the Call Through method, the DoD client application places a call to a specific PBX number. After the call is connected, it dials the pin number (if any required) and then the dialed number. This method can particularly be used if someone calls very frequently into a company PBX or uses a calling card to make calls.

Besides Call Back and Call Through, the application also lets the user to dial calls using the default phone application. Hence, the application does not block the default calling method, but instead gives the user two more ways of making cheaper phone calls.

The third most unique and important feature of DoD client is its ability to operate transparently. With this feature, the user can leave this application running in the background and use his/her phone normally. When making calls, the user does not have to bring the application to foreground to use it. He/she just have to dial calls in the normal way and the DoD client application will intercept the call and dial it again using the method set in the application settings. In the next section we will see how the application settings can be entered or edited.

Another feature of this application is the White List. White List is the list of numbers which are always to be dialed directly. We will see below how this is set up in the application.
APPLICATION USABILITY AND SETTINGS

After the application is installed on the phone, it can be started from the installation location of the mobile phone. Figure 4.3 shows the start screen of the application.

To be able to use the application it has to be configured. If you go into Option->Setup (Figure 4.4), you will see that there are five different settings, i.e. Server Settings, Call Back Settings, Call Through Settings, Operational Settings and White List.

Server Settings page (Figure 4.5) has four mandatory parameters, the Server IP Address, Server Port Number, Username and Password. The Server IP Address will take the IP address of the DoD server, the Server Port Number takes the DoD server port, Username and Password are the username and password of the user using the DoD services.

The Call Back Settings page (Figure 4.6) has three parameters, i.e. Call Back 1, Call Back 2 and Active Call Back. Call Back 1 and Call Back 2 will take two telephone/mobile numbers on which you want to receive the call back from the DoD server when making calls using the Call Back method. Active Call Back tells which one of the two call back numbers, i.e. Call Back 1 and Call Back 2, the user wants to get the call back.

The Call Through Settings page (Figure 4.7) has three parameters, i.e. Access Number, Pin Number and Pause. Access Number is the number of the company PBX or of the calling card. Pin Number is the extension of the PBX or the pin number of the calling card. This is an optional parameter and can also be left empty. Pause is the amount of time in seconds the application waits before dialing the PIN number and after being connected to the Access Number.

The Operational Settings page (Figure 4.8) has only one parameter, i.e. the Call Method, which can take one of four values (Figure 4.9), i.e. Standard Call, Call Back, Call Through and Always Ask. As we will below, this setting determines the default behaviour of the phone.
The last setting page is of White List (Figure 4.10). As mentioned earlier, White List contains the list of numbers that are always to be dialed directly. As you can see on Figure 8, the White List setting page allows adding new numbers to it and editing or deleting the existing ones.

Figure 5.5 Symbian Client Server Settings

Figure 5.6 Symbian Client Call Back Settings

Figure 5.7 Symbian Client Call Through Settings

Figure 5.8 Symbian Client Operational Settings
Once all appropriate settings have been made, the application can be sent to background using its Transparency Mode feature. Enabling Transparent Mode is one of the options available on the main application screen (Figure 4.11). Alternatively, the Back button on the main screen (Figure 4.12) can be pressed to enable the Transparent Mode.

After the application is sent to the background, the user can use the phone in the normal way. When the user tries to make a call in the usual manner, the phone will behave in the way as set in the Operational Settings. If Call Method in Operational Setting was set as Call Back, a request will be
sent to the DoD server for a call back call. Or if it was set to make a Call Through call, the access number in the Call Through settings will be automatically dialed by the application, followed by the pin number and then the actual dialed number. If the user chose Standard Call as its Call Method, then a normal call will be dialed by the application. But if it was set as Always Ask, the user will be prompted with a question asking to select one of the three methods (i.e. Call Back, Call Through or Standard Call) to proceed with the call operation.
6) **Comparisons and Results**

In this chapter we will compare the Java DoD Client with the Symbian DoD Client. The comparison will be based on the requirements that were mentioned in chapter 3. The major purpose of this comparison is to see how well the requirements have been met in the new DoD client.

**User Friendliness**

After analyzing the J2ME DoD client in chapter 3, we realized that incorporating user friendliness was the most important requirement for the new DoD client. In chapter 5 we worked towards implementing an application which is transparent to the user and that does not change the user behavior while making calls and still being able to use the functionality of DoD application.

If we compare the way of placing calls via J2ME DoD client and the Symbian DoD client, a huge difference can be seen. As we mentioned in chapter 3, to make calls from the J2ME DoD client, the client application needs to be started from the applications folder. After the application starts, the number can be dialed on the application screen (Figure 3.2) and then the "Call" button can be pressed to place the call. On the contrary, as mentioned in chapter 5, the Symbian client does not need to be started from the applications folder to place a call. Once the application settings are done correctly and the application is running in transparent mode, a call can be placed in the normal. Depending on the settings done in the Operational Settings, appropriate call method will be selected to place the call. If the call method was selected to be "Always Ask", the user will be prompted to select the call method (Figure 6.1).

![Figure 6.1 User prompt to choose call method when it is set to Always Ask](image)

It is evident from the description above that calls from the Symbian client can be place in a more natural manner and with least amount of change in user behavior than with the J2ME DoD client. Therefore, the first requirement of incorporating user friendliness in the new DoD client is met to a greater extent.
APPLICATION DESIGN

By looking at the class diagrams of both the client applications, that is the J2ME and the Symbian client, it is clearly visible that the design of the Symbian client much more structured and is better divided into logical components than the J2ME client. The class diagram of the Symbian client very well shows that each feature in the application has a corresponding class in the backend. On the contrary, this is not visible from the class diagram of the J2ME client. Also, it is very convenient to extend the Symbian client because it is so well divided into logical components. All these factors correspond that the object oriented design of the Symbian client is better than that of J2ME client.

APPLICATION START TIME

As mentioned in chapter 4, the J2ME applications are supposed to take longer to start in comparison to the Symbian applications. This was further proved when the Symbian DoD Client was compared to the J2ME DoD Client for application start time. The Symbian DoD Client application was starting considerably faster than the J2ME DoD Client.

RUNTIME SPEED OF APPLICATION

An improvement in the runtime speed of the Symbian DoD Client was noticed in comparison to the J2ME DoD Client. The forms and graphics in the Symbian application were being loaded faster in the J2ME application. Also, the requests for call back calls were being sent at a quicker speed in the Symbian application than in the J2ME application. Hence, there was an overall improvement in the application speed.

APPLICATION FEATURES

The Symbian DoD client offers more features than the Java DoD client. Due to the presence of low-level telephony APIs in the Symbian platform, it was possible to implement the call through feature in the application. The same was not possible in J2ME as there were no such APIs to do that. Also, the Symbian DoD client application allowed the users to standard calls even when the application is running, however, in the Java DoD client, the application had to be closed down to make standard calls.
7) **CONCLUSION**

In this chapter, we will conclude this thesis report by pointing out the basic conclusions and major achievements of this thesis work and provide some recommendations for future work.

**ACHIEVEMENTS AND CONCLUSION**

Through this thesis work, several conclusions can be drawn:
- Symbian C++ being native to Symbian OS is better capable of providing access to low level functionalities and System APIs than J2ME.
- Symbian C++ applications have better performance in terms of application start time and run time speed than J2ME applications.
- The Symbian platform is better structured and therefore the application developed on top of it are well structured as well.

**FUTURE WORK**

There exists immense scope for improvement in the project, both on the server and the client side. The server can be enriched with other functionalities like SMS-call back, user presence, follow me and call divert. This will also result in implementing the features in the client application.

The newly developed Symbian client will only cater to a specific market segment. There are other platforms in the market like Windows Mobile, iPhone and Android. Similar client application can be developed to cater to a larger portion of the market.
REFERENCES


http://www.forum.nokia.com/info/sw.nokia.com/id/07ee2527-c07b-4f91-b428-fad5d38ae41f/Symbian_OS_View_Architecture_v1_0_en.pdf.html, 27 May 2005

http://en.wikipedia.org/wiki/Asterisk_(PBX), 18 June 2010


[22] “Developer Platform for Series 60: Using the Core ETel APIs”, Version 1.0, From Forum Nokia, available at:
http://www.forum.nokia.com/info/sw.nokia.com/id/590cac07-c99a-41ad-9f2c-9073c6e6b04/DP_for_Series_60_Using_the_Core_ETel_APIs_v1_0_en.pdf.html, 15 December 2003


[25] “How to send other Application to Background”, From Forum Nokia Wiki, available at:


[40] Ning Zhou, Dial Over Data via GSM Gateway. Master Thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology (KTH), Sweden, June 2005

