A General Logging Service for Symbian based Mobile Phones

RUI GAO

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2007
A General Logging Service for Symbian based Mobile Phones

RUI GAO

Master’s Thesis in Computer Science (30 ECTS credits) at the School of Computer Science and Engineering Royal Institute of Technology year 2007
Supervisor at CSC was Mikael Goldmann
Examiner was Stefan Arnborg

TRITA-CSC-E 2007:132
ISRN-KTH/CSC/E--07/132--SE
ISSN-1653-5715
A General Logging Service for Symbian based Mobile Phones

Abstract

Event logging on mobile phones is interesting for e.g. diary keeping. We present a logging service which – automatically and in the background on Symbian based mobile phones – logs events originated by user interactivities with mobile phones. Context information that can be obtained by the phones themselves is logged as well. The service offers direct access to the logged events, so that data can be retrieved and reviewed. This logging service was developed with aim of logging user activity related data for the Affective Diary project at SICS. The Affective Diary project aims at reflecting user’s emotional experiences. There is a PC-based Affective Diary application developed at SICS. For an Affective Diary application to be able to review the data logged on mobile phones, a middleware service helping to handle the data transferring between mobile phones and PC was also developed. The logging service was designed to be easily reused and extended by third party developers. During the project work we followed a software development process which includes feasibility study, design, implementation and verification. As a result, the logging service ran stably and was reliable.

En generell loggningstjänst för Symbian-baserade mobiltelefoner

Sammanfattning

Preface

This Master’s project was performed at SICS in Kista, Sweden. I would like to thank my supervisor at SICS, Martin Svensson, for the great support that I have received during my work on this Master’s project.

Rui Gao, February 2007, Kista, Sweden
## Contents

1 Introduction ..................................................................................................................... 1  
1.1 Background .............................................................................................................. 1  
1.2 Problem .................................................................................................................. 2  
1.3 Method .................................................................................................................... 3  
1.4 Scope ....................................................................................................................... 4  
1.5 Related Work ......................................................................................................... 5  
1.6 Thesis Overview .................................................................................................. 6  
2 Hardware and Software Environment ............................................................................. 7  
2.1 The P910 Smartphone ............................................................................................ 7  
2.2 The Symbian Operating System ............................................................................. 7  
2.2.1 Constraints ...................................................................................................... 8  
2.2.2 Software Basics ............................................................................................ 9  
2.2.3 Processes, Threads and Multitasking ............................................................. 9  
2.2.4 Descriptor and Exception Handling ............................................................... 11  
2.3 Development Environment ................................................................................... 12  
3 Design and Implementation ........................................................................................... 13  
3.1 Requirements ....................................................................................................... 13  
3.2 What is an Event ................................................................................................ 13  
3.2.1 Symbian OS APIs ..................................................................................... 14  
3.2.2 Events Relevant to Affective Diary ............................................................. 16  
3.3 Event Logging Strategies .................................................................................... 17  
3.4 Data Storage ......................................................................................................... 18  
3.4.1 Stream and Store ...................................................................................... 18  
3.4.2 Database ..................................................................................................... 19  
3.4.3 Temporary Storage of an Event ................................................................. 19  
3.4.4 Caching ....................................................................................................... 20  
3.4.5 Filtering ...................................................................................................... 20  
3.5 Event-Driven System Modeling ............................................................................ 21  
3.6 Event Handling System ....................................................................................... 23  
3.6.1 The Principle of Active Object .................................................................. 23  
3.6.2 Client/Server Framework .......................................................................... 25  
3.6.3 The Client/Server Architecture in the Project ....................................... 27  
3.6.4 Communication Service ........................................................................... 27  
3.7 System Architecture of the Logging Server ...................................................... 29  
3.8 Design Patterns .................................................................................................. 29  
3.9 Problems and Solutions ...................................................................................... 32  
4 Verification and Results ............................................................................................... 36  
4.1 Stability ............................................................................................................... 36  
4.2 Reliability .......................................................................................................... 37  
4.3 Data Integrity .................................................................................................... 38  
5 Discussion and Conclusions ....................................................................................... 39  
5.1 Discussion .......................................................................................................... 39
5.2 Conclusions ....................................................................................................... 40
5.3 Future Work ..................................................................................................... 40
References ............................................................................................................... 42
1 Introduction

Within the last few years, mobile phones have been widely used, so that they become a common instrument taking part in our daily lives. They are almost always on and tend to have an intimate connection with their users. They record the events originated by the users, for instance the phone calls the user made, the SMS messages sent or received, and the photos taken etc. They may even obtain knowledge about the surroundings, such as near-by Bluetooth presence. All these kinds of information may be interesting to log for personal reasons such as diary keeping or even for medical reasons where the information may be used for analysing a person’s behaviour. Having a programmable platform, the mobile phones provide software developers possibilities to create applications for accessing this kind of information on the mobile phones. In this paper we introduce a logging service developed on the mobile phone platform that logs information which we were interested in and offers access to the logged data.

1.1 Background

A diary is generally considered to be a book used for a personal record of events, experiences and observations. We may combine words with personal jottings, song lyrics, drawings etc in our diaries to form our ways of expression and re-livings. However, there are various other media that might be considered to be used in diary-keeping [9]. Smartphones provide access to media such as e-mail, text messaging, audio recordings, as well as photos, which we might come to associate with experiences or people. Using sensor-based technologies such as biosensors, it is also possible to obtain bodily aspects of emotional experience in our daily lives, e.g. pulse, skin temperature, movement, and other autonomic reactions. All these pieces of information might be incorporated into a diary.

Affective Diary is a project conducted at the Interaction Lab at SICS (Swedish Institute of Computer Science) in cooperation with Microsoft Research. The purpose of the Affective Diary is to provide users with a way to remember their emotional experiences [4, 9], and create a representation that they can later interpret and reflect on. In Affective Diary, the system will on the one hand make use of mobile phones and biosensors to collect data, and on the other hand allow for the ongoing processes of interpretation and reflection.

The Affective Diary consists of a mobile phone, body sensors, and a Tablet PC (see Figure 1.1). By activating the Affective Diary logging service on the mobile phone, and the body sensor armband, a user starts creating her Affective diary. During the day, the system collects activities on the mobile phone: sent and received SMSs/MMSs, photographs, and
Bluetooth presence; and biosensor data: movement and galvanic skin response (GSR\(^1\)). Once the user is back at home she can transfer the logged data onto her Tablet PC-based Affective Diary application.

![Affective Diary system setup](image)

In the Affective Diary application the collected sensor data are represented as an anthropomorphic, abstract character with different postures depending on the user’s level of physical activity and coloured depending on the user’s level of excitement [4, 9]. The data from the mobile phone are placed above the character (see Figure 1.1). All the data are arranged along a timeline and the representation of them can be played as a movie. The Affective Diary also allows the user to work with, alter, and scribble on top of the collected data to piece together her own story.

### 1.2 Problem

At SICS, a prototype of Affective Diary has been developed. This first prototype was not completely stable. Without question, the two most difficult problems were securely and autonomously collecting mobile phone events and automatically transferring those to the Affective Diary application. In fact, in the developed prototype a user had to constantly check the logging software on the phone as well as manually transfer data between the phone and the PC. Consequently, the main goal of this Master’s project was to design and implement a software program that automatically, and in the background, captures events relevant to the Affective Diary, and to provide an application program interface (API) that allows access to the events logged on mobile phones. Furthermore, the project also aimed to provide a way to automatically transfer information between the software on a mobile phone and a PC-based Affective Diary application. The software program should be reliable, stable, reusable and extendible and should result in a general logging service.

\(^1\)GSR is a method to understand various types of activity in certain parts of the body.
In this thesis we describe events as those that are initiated by user activities on a mobile phone and could reflect a user’s emotions and behaviours. Each type of events will be handled by a corresponding event handler in the software program. The entire event logs can later in PC applications to be reviewed and reflected on by end-users as a diary.

There were two general questions this Master’s project tackled:
1. Given the Symbian public APIs [13] and third party software¹, which events (SMS, incoming/outgoing phone calls, MMS, and so on) are possible to log?
2. Which events are interesting to log for an Affective diary?

This Master’s project focused on the difficulties in capturing and “storing” events and to transfer data between a mobile phone and a PC application. Therefore, we wanted to answer the following specific questions:
3. How to capture and log information on mobile phones?
4. How to handle large amounts of data on mobile phones? A potential problem is that the captured information may be too large to store. How can information be filtered to avoid logging redundant information?
5. How to minimize system resources? A service should always be running in the background. It is vital that it does not disrupt a user’s normal interaction with the phone in terms of consuming processing power and reducing battery life.
6. How to automatically transfer information between the software on a mobile phone and a PC-based Affective Diary application?

Although this project was focused on the Affective Diary application, we wanted to design and implement a mobile phone logging service that can be used in other application scenarios, and can be reused and further developed. Thus, the final questions were:
7. How to create a general event logging API?
8. How to design the software program so that it results in a framework which is reusable, extendible and portable?

1.3 Method

Given the goals it is clear that this Master’s project was more technical than theoretical, thus our chosen method should reflect that. The project consisted of three steps: feasibility study, design/implementation and lastly a verification of the developed software.

Feasibility study
This Master’s project was developed on the Symbian platform in C++. To be able to start, it was necessary to get sufficient knowledge about the Symbian OS, its C++ coding standards and conventions. Another important aspect was to understand the extensive API. With that knowledge at hand, some investigations into how to deal with the problems described above

¹ E.g. Sony Ericsson MMS API and Mobinfo API (http://developer.sonyericsson.com)
were carried out. These included finding events that can be logged based on the Symbian OS APIs and third-party APIs, investigating relevant literature on event logging methods, studying related work, and researching the related topics of event-handling and client-server communication.

**Design and implementation**
The design of the software program was performed using the Unified Modeling Language (UML). Several design patterns [5] were used to achieve the goal that the software program results in a framework which is reusable, extendible and portable.

The implementation was performed by setting up several development milestones:
1. A preliminary software prototype that defines a logging API providing access to a database where the captured events are stored, and that has a simple event handler implemented which triggers when a relevant event occurs (e.g. SMS comes in) and logs the captured event.
2. Complete the software program with several event handlers depending on which types of events are interesting to log.
3. An application that on the mobile phone side automatically handles information transfer between the software on a mobile phone and a PC application.

**Verification and Evaluation**
After the completion of the entire program, the software was verified by several verification tests:
1. To ensure stability, two test users performed a verification test by running the software for a period of at least 30 days, and reporting any disturbances or system crashes the software caused.
2. A detailed consistency verification test was also carried out. A test user, in the course of two days, verified that the software logged events correctly, by using a given predefined set of events.
3. Finally, data transfer between the software on a mobile phone and a PC-based Affective Diary application was verified by connecting these two devices and controlling the data integrity.

Conclusions could be drawn after studying the results acquired from the tests: the logging service ran stably. It did not disrupt a user’s normal use of the mobile phone. It worked correctly and was reliable. No data were lost during any transfer.

### 1.4 Scope
This Master’s project focused on developing a logging service on the Symbian platform that will automatically log information of events, and allow them to be transferred to a PC-based Affective Diary application. Such events may be generated by a number of different Symbian OS components. Due to the current Affective Diary application, few of them were
handled in this project: incoming/outgoing SMS messages and phone calls, Bluetooth presence, and photo taking.

The project was designed to handle data filtering and caching. Since we were not focusing on investigating advanced filtering strategies and caching algorithms, only some simple filtering strategies to avoid logging redundant data were implemented.

For transferring data between the software on a mobile phone and a PC application we chose to develop a simple communication service that exchanges information by writing to and reading from command files, while other connectivity methods, such as Bluetooth, are also supported by the devices.

In the process of project performance, the developed software was only verified continuously on two SonyEricsson Smartphones by two users.

1.5 Related Work

Over a few years, mobile phones have become common devices in our everyday life. Much research into how to log users’ activities and their environments using mobile phones has been done.

The Department of Computer Science at University of Helsinki and the Research Unit at Helsinki Institute for Information Technology have developed a software platform for context-aware mobile applications named Contextphone, which is available as open source (http://www.cs.helsinki.fi/group/context/). ContextPhone consists of four interconnected main modules [10]:

- **Sensors**, being used to acquire context data from different sources.
- **Communications**, providing connections to external services via standard Internet protocols.
- **System services**, being used for automatic starting background services, error logging as well as recovery.
- **Customizable applications**.

The software works on Nokia Series 60 Smartphones running Symbian OS (see Section 2.2). ContextLogger [10] is one of the customizable applications provided by Contextphone. It is developed using the Sensors for recording mobility data. ContextLogger receives notifications of context changes, writes these data in a local file, and periodically uploads the files to the researcher’s server via the background file upload [10]. Many research groups were using the data collected by ContextLogger to reflect on the users’ daily lives, e.g. the MIT Media Lab’s Reality Mining group (http://reality.media.mit.edu) used ContextLogger to discover regular rules and structure in the behaviour of both individuals and organizations [3]. The Technical Research Centre of Finland (VTT) was using ContextLogger for modelling the recurring patterns of a work group’s person-to-person interactions [15].
Symbian provides a Log Engine API [13]. This API is used to record events of interest to the user. These events can be stored in a database, and later be retrieved by a viewer application and displayed to the user. The Log Engine API is extendible in the sense that other applications can add new event types through the API. Currently available event types handled by the Log Engine are phone calls, fax, SMS, and e-mail events.

In this Master’s project, the developed software acts not only as a logging service running in the background, but also offers different ways of retrieving logged events: through recreating logged events as well as generating XML files. Furthermore, the software provides automatic synchronization between a phone and a PC application when a connection has been established. In the implementation, we did not use the Log Engine API for logging and retrieving events, since we found that the CLogEvent class provided by Log Engine tended to only handle events related to user interaction with the phone, but the events we wanted to handle may include context events. Thus the Log Engine API is not as general as we needed. Furthermore, it is important for us that events can be retrieved over time, but this is not provided by the Log Engine API. We could not make use of the Contextphone open source either, since the Contextphone software works only on Nokia Series 60 Smartphones, and the main target mobile phones in our case are the SonyEricsson Smartphones. Furthermore, the ContextLogger application logs events in log files, but we wanted to be able to recreate logged events rather than only generate plain log files.

1.6 Thesis Overview

This thesis is organized as follows:

Chapter 2 – Hardware and Software Environment contains a description of the hardware and software environment where the software program developed in this Master’s project runs.

Chapter 3 – Design and Implementation describes relevant aspects of the Symbian OS and theories that support the design decisions we made in different situations, and specifies what we want to build in this project and how it can be built. We will also describe the different pieces of software developed in this project, in particular, the problems we met during the development, and the respective solutions.

Chapter 4 – Verification and Results describes the methods used for verifying the developed software and the most important results.

Chapter 5 – Discussion and Conclusions contains a discussion regarding the developed software, conclusions made from the project work and suggestions for future improvements.
2 Hardware and Software Environment

In this chapter we describe the hardware and software environment in which the developed software service runs. We take a look at the features of the target Smartphone – particularly the features that we are interested in – and introduce the Symbian OS.

2.1 The P910 Smartphone

The Sony Ericsson P910 Smartphone was the main target mobile phone for this Master’s project. The P910 includes, for example, PDA functionality, a still and video camera, e-mail and a web browser. The P910 supports GSM/GPRS, Bluetooth, infrared and USB connectivity. The phone has an internal user memory of 64 MB, and supports 1 GB of external memory, which makes it ready for external application use. The P910 is based on Symbian OS v7.0, and uses the UIQ\(^1\) v2.1 user interface. It is backward-compatible with the P800 and P900 Smartphones.

2.2 The Symbian Operating System

Having its roots in Psion Software’s EPOC\(^2\), Symbian OS is structured like many PC operating systems, with process management, pre-emptive multitasking (see Section 2.2.3), memory protection, file management and advanced graphical user interface management. Symbian OS is also very different from PC operating systems. It is a real-time OS where resources are constrained and power management is critical [6]. An application developer has to conserve memory and tackle errors such as memory leaks, since the Symbian OS may be running for months or even years without rebooting. Symbian is written in C++, with an object-oriented design. It is therefore natural to develop applications in C++, however, many Symbian devices can also be programmed in Visual Basic, Perl, Java etc.

The following description of Symbian OS and its programming model in this chapter is primarily based on [6] if not otherwise noted.

---

1 UIQ is a user interface and development platform that is used on Symbian OS. www.uiq.com
2 EPOC is a family of operating systems developed by Psion primarily for PDAs
2.2.1 Constraints

As mentioned above the resources in a mobile phone are much more limited than those of a typical PC. Hardware resources are limited, which causes many restrictions on software and software development. The main rechargeable battery is the only power source, which provides much lower power than that used by PCs.

- Symbian OS systems are currently based on 100 MHz to 210 MHz ARM9\(^1\) processors. The most recently released SonyEricsson P990 has a processor of 208 MHz, and the P800, P900 and P910 which were released a couple of years ago have a processor of 156 MHz. This is much slower compared with the processor in PC computers.
- There is no hard drive; the user memory and the supported external memory are limited. We cannot suppose there is an infinite amount of space to store our program or data files.
- The read-only memory (ROM) contains the OS and all the built-in middleware and applications. The ROM is accessible in the sense that it consists entirely of files in a directory tree. Symbian programs in ROM can be run directly in-place, while PC programs have to be copied to RAM and then execute. Typical Symbian OS v7.0 machines use around 20 MB of ROM.
- The random access memory (RAM) is used by active programs and the system kernel. The total RAM on a typical mobile phone is only around 16 MB (SonyEricsson P910 has a RAM size of 64 MB). With a RAM of small size, it is possible that the mobile phone may get exhausted and fall in out-of-memory errors. Therefore, current mobile phones use flash RAM more often.
- Each thread has a very small stack and a default size heap. If a thread overflows its stack, the program may crash and the Symbian OS has to be rebooted. This means only small object can be put on the stack, all large objects should be kept on the heap. The usual initial stack size is only 12 KB, and the maximum heap size on P910 is 8 MB.
- Power management is possibly what differs most between a PC and a mobile phone. Mobile phones are expected to work for several days between recharges. Thus, power has to be used efficiently. For example, the Symbain OS saves power by switching CPU off when all threads are idle.

Thus, programs developed on Symbian have to work efficiently making the best use of the scarce resources. Fortunately Symbian is designed from the ground up to help software developers to do this. The following sections introduce the fundamentals of the Symbian OS and some typical Symbian specific programming idioms to deal with the memory and power management.

---

\(^1\) The Advanced RISC Machine version 9 (ARM9) is a 32-bit RISC processor architecture that is widely used in a number of embedded designs. ARM processors are dominant in the mobile electronics market due to their power saving features.
2.2.2 Software Basics

There are different types of software that need to run on Symbian: kernel, servers and user applications.

The kernel manages the mobile phone’s hardware resources such as system RAM and hardware devices. All other software components can only access the system resources through the kernel. The kernel of Symbian has a micro architecture. It manages memory and schedules processor time for each process, but it does not support networking or file system management. These things are provided by servers.

A server is a program that manages one or more related resources. Each server provides an API so that clients can gain access to the services. Two commonly used servers in Symbian are the file server, which handles all files, and the window server, which handles user input and drawing to the screen.

A user application is a program with a user interface. A typical user application in Symbian is divided into an engine part and a graphical user interface (GUI) part. The engine manipulates data and the GUI interacts directly with the user. An application engine may be a separate source module or a separate pre-compiled library.

The software we wanted to develop in this project manages event logging and provides client access to the logged events. Thus according to the Symbian convention we call this software a logging server.

2.2.3 Processes, Threads and Multitasking

The process is a way to group related resources together. It is a fundamental unit of protection in Symbian. Each process has a private memory address space containing program text and data. The virtual addresses used by programs executing are mapped onto the physical addresses by the system’s memory management unit (MMU).

The thread is a fundamental unit of execution. There are one or more threads per process. Different threads in a process execute independently of the others, but have the same address space. Threads are the actual entities that are scheduled for execution on the central processing unit (CPU).

Multitasking is a method by which multiple tasks share common processing resources such as a CPU [16]. With one CPU, only one task is executing at any point in time, multiple tasks have to be scheduled for execution on the CPU so that each task gets its own processing time. The act of switching the execution from one task to another is called a context switch. The most costly type of context switch is between two threads in different processes. Switching contexts in the same process consumes far less CPU resources. There are two
methods to share processing time with all running tasks in multitasking, pre-emptive multitasking and non-pre-emptive multitasking. In pre-emptive multitasking each running task gets a recurring slice of processing time from the CPU. When important external events, e.g. incoming data occur, the operating system can pre-empt or stop a currently running task for a higher priority task of data transfer. In non-pre-emptive multitasking the operating system never originates a context switch from a running task to another. The running task does not release the CPU until it becomes blocked, e.g. when waiting for another task, or until it voluntarily gives control to other tasks. Non-pre-emptive multitasking is often used when tasks are designed to cooperate with each other in order to work efficiently.

Symbian OS implements thread based multitasking, also known as multithreading, pre-emptively, so that it can run multiple applications and servers simultaneously. In Symbian, there is a fundamental idiom called active object (see Section 3.6.1), which is used to achieve non-pre-emptive multitasking within a single thread. Typically, each Symbian application and server uses only one thread per process and handles multiple tasks using active objects. The benefit of using a separate process for each application and server is increased memory protection – a process can not accidentally access and write to other processes’ memory. In the cases where a few servers must work together closely, they can sometimes be packaged into a single process to reduce the cost of context switching.

![Diagram of processes, threads and active objects](image.png)

Figure 2.1 illustrates two processes. Each process operates on its own address space and has a single thread of control. Each of the threads contains several active objects managing non-pre-emptive multitasking within the context of the thread. Multithreading is managed pre-emptively by the Symbian OS kernel.
2.2.4 Descriptor and Exception Handling

When the Symbian OS was developed a number of years ago, the resources on a mobile phone were even scarcer. It was vital to find a safe and reliable way for containing, accessing and manipulating data. Thus, Symbian uses descriptors with pointers and length information to describe strings and binary buffers instead of using e.g. C++ strings or buffers. In this way, any attempts to access memory outside the defined data area can be caught. There are different types of descriptors provided in Symbian: buffer descriptors (TBufC and TBuf), pointer descriptors (TPtrC and TPtr) and heap descriptors (HBufC). Figure 2.2 illustrates how data are organized within these descriptors.

Buffer descriptors include the data as a part of the descriptor, while pointer descriptors and heap descriptors have their data stored separately from themselves. The descriptors use the length information to specify the size of the data, and may also include a maximum length to describe the total size of the allocated memory.

The well-known `try/catch` keywords for exception handling are not provided in Symbian. Instead, exceptions are handled by the Symbian `leave` and `trap`. A `leave` causes execution of the active function to terminate, and on through all calling functions, until the first `trap` is met. We could assume that the `leave` is like `throw`, and the `trap` is like `catch`. A function, which may throw exceptions, is called a leaving function, and has a trailing “L” in its name, for example, the `NewL()` function which defines a constructor that can throw an exception.
A cleanup stack mechanism is used in Symbian to prevent objects that have been allocated on the heap from becoming orphaned if an exception occurs, particularly in the cases where an object may lose its reference during the object instantiation and memory allocation. Thus, every object allocated on the heap should be protected by pushing its pointer to the cleanup stack. With the cleanup stack, when a leave occurs, the associated trap will call a Pop() function over all the currently allocated objects held on the cleanup stack to pop and destroy them. A typical usage of the cleanup stack is shown in the following code.

```cpp
CX* x = new (ELeave) CX;
CleanupStack::PushL(x);
x->UseL();
CleanupStack::PopAndDestroy(x);
```

When a new object of CX has been instantiated, an amount of memory is allocated on the heap and a pointer x pointing to the object is created on the current stack. The code pushes the pointer x to the cleanup stack and calls the UseL() function. If a leave occurs at that time, the stack frame on the current stack, which contains x, may be discarded without deleting the object. Anyway, the object is destroyed because of the cleanup stack mechanism. If the UseL() does not fail, the code pops the pointer x from the cleanup stack and deletes the object.

### 2.3 Development Environment

To develop in Symbian OS C++, we chose the Carbide Development Tools from Nokia (http://www.forum.nokia.com). Carbide is based on the open Eclipse framework (http://www.eclipse.org), and supports Symbian development on Nokia Series 60 and UIQ platforms. Since we had experience working in an Eclipse integrated development environment (IDE), it was natural to start working in Carbide. The development environment we used was the Carbide C++ Express v1.0, which is a free IDE provided by Nokia.

To get the Symbian developer libraries, header files and utilities for building applications in Symbian OS C++, we installed the Symbian OS v7.0 UIQ 2.1 software development kit (SDK) which is available as free download from SonyEricsson (http://developer.sonyericsson.com). The SDK includes a UIQ Emulator, hosted on a Windows-based PC, on which programs can be debugged. However, we could not make use of the UIQ Emulator, since the logging server we wanted to build runs in the background, and all events we wanted to log are generated on an actual mobile phone. Even if we could simulate all kinds of events, there was no way to access a phone file system by the emulator and create a database for keeping the events. Furthermore on-device debugging was not available in the version of Carbide we had access to. Thus, we chose using text files containing print-outs from the logging server as the debug method.
3 Design and Implementation

This project set out to tackle a number of problems with respect to event logging on mobile phones such as: What events can be captured? What events are interesting to log? How to keep power consumption at a minimum? This chapter describes the various APIs in Symbian (and also other third party software) for capturing events as well as the design decisions that were made in order to create a logging server that logs relevant information and adheres to Symbian OS programming conventions. We analyse the behaviour of the logging server we wanted to build and specify different components and relationships between them. We will also take a look at how general design patterns help ensure program reusability and generality. Finally, we describe the problems we met during the development, and the respective solutions.

3.1 Requirements

The requirements on the design were that the logging server should result in a program that is reusable, extendible and portable. That is to say, the code should be easily reused by third party developers. New events can be handled by extending the logging server with new event handlers. The logging server can be ported to other platforms, such as P900, without difficulties. Furthermore, the logging server should be reliable, run stably in the background, and work correctly. The logged events should also be maintained, e.g. no data should be lost during any operation.

3.2 What is an Event

Events considered for logging in this project are those that are originated by user activities on a mobile phone. Users’ daily interactions with their mobile phones initiate such events. Activities such as dialling/answering phone calls, sending/receiving messages and taking photos trigger events. Furthermore, context changes trigger events on the phone as well. The mobile phone exchanges information with nearby Radio Base Stations (RBS) continuously, each piece of the sent/received information from the RBSs can be seen as an event. With enabled Bluetooth, the mobile phone can detect other Bluetooth devices’ movements, which can also be considered as events. These events are not triggered by the user directly but can still reflect the user’s activities such as movements. However, the events considered here do not have anything to do with the system events that represent health status of the system.

Based on the Symbian public APIs and third party software, a number of events may be
captured. In the next section we take a look at different Symbian OS components, and examine which events are possible to log, which are interesting to log for an Affective diary, and which are chosen to implement in this project.

### 3.2.1 Symbian OS APIs

Bluetooth, File system, Messaging, and Telephony are the services on a mobile phone that are involved in what we mean can reflect user activities and initiate events. Symbian provide public APIs to access them.

**Bluetooth**

Bluetooth provides a standard for short-range wireless connections between computers, Pocket PCs, mobile phones and other devices [1, 13]. The Service Discovery Protocol (SDP) enables devices to search for and discover other devices and the services they offer over Bluetooth. Since the Bluetooth radio signal has a short range of 10 meters, the Bluetooth devices can come in and out of range of other devices easily and quickly. Device discovery is a way to detect this movement. Each Bluetooth device has a unique 48-bit address to identify itself; and each Bluetooth service is specified by a service ID number. Besides the ID number, a service record, which consists entirely of a list of service attributes, describes which services the Bluetooth device offers. This service record is stored in a local database maintained on Bluetooth devices [13].

The Symbian OS provides APIs to discover Bluetooth devices and the services they offer: the Bluetooth Socket API is used to query for available remote devices through a Host-Resolver socket, and the Service Discovery Agent (SDA) uses the Bluetooth Service Discovery Protocol (SDP) to search for available services on the remote device [13]. Thus, the interesting information we can obtain from the Bluetooth APIs are the names and addresses of the remote devices, as well as services offered on each of them. Every discovered remote Bluetooth device can be considered as an event to log. Including the name, the address and the offered services in the event log, we can identify the device as well as describe the device’s characteristics in detail.

**File systems**

Symbian provides APIs for accessing file systems [13]. The functionalities provided are file system manipulation functions, including adding, removing and moving files and directories, requesting notifications of when a significant change occurs, and obtaining information of drives and volumes etc.

These APIs offer possibilities for us to observe events occurring in the file system. By observing the contents of a certain folder and requesting notifications of when changes occur, we can for instance be notified when a new photo is created from the camera, and acquire the file path of the photo. This piece of information can be logged as an event indicating that photos have been taken.
**Messaging**

The Messaging framework in Symbian OS provides core messaging functionality and supports extendible multi-protocol messaging\(^1\) [13]. Messaging types in Symbian OS are defined by Message Type Modules (MTMs) [13]. Each MTM is a software plug-in that is responsible for handling a single message type, such as SMS text messages. Thus, the messaging functionality of Symbian OS can be extended by implementing new MTMs. The currently supported MTMs in the standard UIQ 2.1 SDK are [13]:

- Fax MTM, handling fax transports.
- IMAP4\(^2\) MTM, providing retrieval and management of Internet e-mail on a remote IMAP4 server.
- POP3\(^3\) MTM, allowing Internet e-mail to be retrieved using the POP3 protocol.
- SMTP\(^4\) MTM, providing sending of Internet e-mail messages using SMTP.
- SMS MTM, providing sending and retrieving of SMS text messages.
- MMS MTM, providing sending and retrieving of MMS messages.

Symbian offers APIs for each of these MTMs.

The SonyEricsson P910 mobile phone has its own version of the MMS API [12], which allows client applications to handle MMS messages on the phone. The Sony Ericsson MMS API is supplied as an addition to the standard UIQ 2.1 SDK, and is based on the Nokia Series 60 MMS API (http://www.forum.nokia.com).

At the time when messages are sent or received, we can acquire interesting information such as the remote party which issued the message, the subject and the message body using the corresponding messaging API. Those pieces of information obtained from a message can be included in an event to log. By logging various types of message events, we can obtain some knowledge of communications on the phone originated by the user.

**Telephony**

A generic interface, ETel Core API, is provided by Symbian to initiate, control and terminate telephone calls [13]. In the ETel Core, a *phone* abstracts a particular telephone device. A *phone* can have one or several *lines*; and a *line* can have zero or more active *calls*. The ETel Core interface allows client applications to access the status and capabilities information of the *phone* device, *lines*, and active *calls*, and can be notified if these change.

---

\(^1\) Multi-protocol messaging refers to process communications that use different protocols depending on where the processes are located and what type of server system is used.

\(^2\) Internet Message Access Protocol version 4 is a protocol for e-mail clients to access and manage mailboxes on a mail server.

\(^3\) Post Office Protocol version 3 is used to provide simple remote access to mailboxes.

\(^4\) Simple Mail Transfer Protocol is a TCP/IP protocol used in sending and receiving e-mail.
Based on ETel, the Mobinfo API is developed by the Symbian Company to provide an easy way to access to the most important telephony properties and services on SonyEricsson’s P800, P900 and P910 Smartphones [11]. The following information can be accessed by the Mobinfo API:

- International Mobile Equipment Identifier (IMEI), International Mobile Subscriber Identifier (IMSI) and own phone numbers
- Network availability and change notification
- Network and location identification, such as GSM cell ID
- Battery and signal strength notification

These APIs make it possible for us to be notified at for instance incoming/outgoing phone calls and network changes. Interesting information such as the remote party telephone number of a phone call and its direction as well as the cell ID of the current network can be obtained and logged as events.

As we have seen, there are numerous events possible to log on the P910 phone using Symbian OS APIs and third party software.

- Using the Bluetooth APIs we can log events describing nearby Bluetooth devices and their respective services.
- Using file system APIs we can log events of file system changes, such as adding/removing photos.
- Using Messaging APIs we can log events of sent/received SMS messages, MMS messages, faxes, and Internet e-mails.
- Using ETel Core APIs and the Mobinfo API we can capture events of incoming/outgoing phone calls, the battery level changes, signal strength level changes, and information about the current network such as the cell IDs.

### 3.2.2 Events Relevant to Affective Diary

However, not all the events that could be logged were interesting to the Affective Diary. A restriction was made that we would only focus on logging the events that could reflect the user’s emotions. For example, SMS/MMS message conversations and frequency of phone calls might show the user’s feelings or relationships to friends, the nearby Bluetooth devices might show the environments the user has visited, e.g. at work or in a crowd, the photos taken might show the people the user has met, and the GSM cell ID information might show geographically where the user has been during a day.

In this project we decided to implement the following events: incoming/outgoing SMS messages and phone calls, Bluetooth presence, and photo taking. These events were chosen because most of them were already handled in the current version of the PC-based Affective

---

1The identification number of an area which covered by a Radio Base Station
Diary application. The phone call event was not handled in the application yet but it was interesting to log for further development of the Affective Diary project. For the same reason we will also discuss other interesting events, such as MMS and GSM cell ID information events.

3.3 Event Logging Strategies

To our knowledge not much research has been conducted in various ways of logging events. In principal there are two techniques that the logging server can use: either store events in text based log files (commonly used by operating systems such as Linux) or store events in a database. Both alternatives have strengths and weaknesses.

Logging events as single-line textual messages is a simple way of event logging. ContextLogger [10] logs events using this method. The benefit of a text based approach is apparent. It is easy to do and often portable as all information is stored in text files ready to be read. On the other hand, text based event logs are not very dynamic and they are difficult to manipulate. For instance, it is difficult to log events that are very different in terms of what they contain and one often resorts to a common format leaving out event specific information. Thus, text based log files are suited for scenarios in which you want to track the behaviour of a system. Since most event logs are text based there are approaches to deal with these shortcomings, specifically data clustering algorithms for mining frequent patterns in event logs. One such algorithm is described in [14]. The algorithm builds clusters of frequent words from event logs, which works as follows. It first passes over the data and builds a data summary. The summary is a collection of all frequent words identified from the log. The algorithm then builds a cluster candidate table by making a second pass over the data set line by line. When one or more frequent words have been detected in a line, a cluster candidate is formed using these frequent words as its attributes. If the cluster candidate is not present in the candidate table, it will be inserted with a support value 1, otherwise its support value will be increased. After all cluster candidates have been built, the support values of the cluster candidates are examined and the clusters candidates that are frequently used are selected by the algorithm as clusters. Generally, this algorithm helps to manipulate event logs by clustering data. In the case of the Symbian platform where resources are extremely scarce, the algorithm turns out to be too expensive, considering the data summarization and data set traversing. Furthermore, the algorithm can only be applied to log files after the logging process, but we prefer ongoing processes of event access.

Another event logging method is using databases. Log Engine [13] uses databases to store events. Symbain provides a database management system (DBMS) API, by which databases can be easily managed on mobile phones. The Structured Query Language (SQL) is supported by this API, so that the logged events can be quickly and efficiently retrieved in different ways from the database. Another advantage of using databases is that different types of events can be logged in the database in one strict format by using one database table, which makes manipulation easier. However the database is not as portable as text files,
it can not simply be moved to other platforms and be managed in the same way as on the Symbian platform. Furthermore, databases may require space overhead. Besides the data, the underlying row storages and table storages must be represented.

Comparing these two event logging strategies, we have chosen to use databases to log events in this project, since it is important to us to be able to manipulate data easily and effectively. The algorithm for mining patterns from event logs we described before could be used on the event logs at the PC side for analysis reasons, when file based event logs have been generated and transferred to the PC side.

### 3.4 Data Storage

Since the logging server should run continuously in a mobile phone, it will collect large amounts of data. In this section, we introduce how data are stored on a Symbian based mobile phone, and describe how the event database was modelled to be as general as possible allowing for insertion of future event types. We will also describe the mechanisms considered for temporary storage of events, improvement of event access performance as well as the strategies for avoiding logging redundant information.

#### 3.4.1 Stream and Store

In Symbian, data can be read from and written to storage as *streams*. A *stream* is a sequence of bytes which is used to represent the external formats of objects [13]. A *store* is a collection of *streams*, and provides persistence of data. Each stream is assigned to an ID number at creation in a store. Given a stream ID, a particular stream can be closed and later reopened again. A *store* can be transient or persistent. A persistent *store* provides the concept of a *root stream*, see Figure 3.1.

![Figure 3.1 A persistent store with a root stream][1]

The *root stream* is accessible from the outside world and is the starting point from which all others *streams* within the *store* can be found. The persistent *store* can be closed and reopened, and still provide access to the same data. Typically, a file can be seen as a persistent *store*, and a database is in fact a permanent file.
3.4.2 Database

There are some limits to be considered when using DBMS in Symbian. The number of rows in a table and the number of tables in a database are limited by the available resources, and the number of columns in a table is usually limited by a maximum record size of 8200 bytes [13]. For instance, a table could have maximal 32 columns, where each column contains 255 bytes data.

We considered two possible ways to create the database tables in this project: creating one table for storing all events, alternatively creating several tables for different event types. We decided to create only one table in the database, since events were expected to be logged and retrieved over occurrence time, the one table strategy offers a natural way to achieve this. Furthermore, in this way the database offers exactly one format for all kinds of events, new types of events can be easily inserted into the same database table without modifying it, which makes the database general for logging.

As we have seen in Section 3.2.1, all the interesting events we wanted to log have some common behaviour. Each of them has an event type, a remote party indicating the one who issued the event, an occurrence time, and other specific information about the event. To model a common format for all kinds of events, we decided to include the following columns into the database table:

- **ID**, a 64 bit integer indicating the id of the event
- **Type**, a text string describing the type of the event
- **Time**, a 64 bit integer timestamp of event occurrence
- **Remote Party**, a text string indicating the name of the remote party issuing the event
- **Description**, a text string describing the event
- **Event Body**, a sequence of binary data representing the EventBody object of the event (see Section 3.4.3)

3.4.3 Temporary Storage of an Event

For temporary storage of an event, we create an Event object. The Event class used for object instantiation was defined containing exactly the same attributes as the columns in the database table, i.e. **ID**, **Type**, **Time**, **Remote Party**, **Description** and **EventBody**, where the EventBody is an object belonging to an Event that describes the specific characteristics of the event. For instance, a Bluetooth EventBody may contain the name and the address of the device and a list of names of available services, and a SMS EventBody may contain the message subject and body. As mentioned in Section 3.4.1 objects can be easily read from and written to storage as a sequence of bytes (a stream). In this way, the specific information kept in the EventBody objects can be read from and written to the respective Event Body fields in the database table without difficulty.
3.4.4 Caching

With the aim of increasing performance of event access, we wanted a database cache. The meaning of the cache was to provide rapid access to recently used events. The cache should have a fixed size and use cache algorithms for swapping out aged events. Typical cache algorithms are Least Recently Used algorithm, which discards the least recently used items first, Least Frequently Used algorithm, which keeps track of how often an item is needed and discards those used least often first, and Adaptive Replacement Cache algorithm, which balances between Least Recently Used and Least Frequently Used algorithms [16]. However we found that each thread running on the SonyEricsson P910 has a default heap size of 8 MB. When the heap reaches its maximum and no other memory is available to be allocated, the system may crash by out-of-memory faults. Allocating an amount of memory on the heap for cache usage may raise the risk of out-of-memory occurrences. Therefore we decided not to implement the database cache.

3.4.5 Filtering

The amount of data that can be stored in the database is limited by the available resources on a mobile phone. Thus, it is essential to avoid logging redundant information. Events have to be filtered. In particular, those events containing repeated information that are captured periodically should be removed.

In our case, this problem became apparent for two event types: nearby Bluetooth device events logging and GSM cell ID events logging. The other relevant Affective Diary events do not have the same behaviour of frequent repetitive information.

Nearby Bluetooth devices are discovered by the logging server at regular intervals. A potential problem with nearby Bluetooth device events logging is that Bluetooth devices may stay nearby for a long time period and hence be unnecessarily discovered many times. For instance in an office environment, in fact Bluetooth printers or mobile phone devices may stay nearby all day long, rendering a potentially huge event database. The solution is simple. Let the logging server log an event when the Bluetooth device first appears, and similarly log an event when the device disappears from the range of Bluetooth discovering ability. In this way, we can describe the existence of the device.

A GSM mobile phone communicates with a base station over the air. For any given location there may be a number of base stations whose radio signals reach the mobile phone. The mobile phone chooses the strongest one, and switches to a new base station when needed. A cell is an area covered by a base station. A base station can have control over several cells. The cell ID we have talked about is the identification number of an area under its corresponding base station. Since cellular networks are almost present everywhere, and cell IDs can be easily obtained, it could be interesting to log cell IDs and translate them into names of physical locations for reflecting user’s movements in Affective Diary. However,
there are some difficulties when logging cell IDs: different cells covered by different base stations may overlap.

![Figure 3.2 An example of cell overlap](image)

As illustrated in Figure 3.2, in the overlapping cell area, if approximately the same radio signal strength reaches the mobile phone, the phone may hop between the cells even when the user is not moving. Furthermore, standing on the boundary of two adjoining cells, back and forth movement by the user may also cause the same phenomenon. Thus, when logging cell ID events, we should filter cell ID information and log information only when the user is actually moving. Laasonen et al. [8] present an algorithm dealing with such oscillations. A cell cluster is defined in [8] as a group of nearby cells where most transitions happen within the cluster. When a cell is shared by several clusters, it is unclear which of these clusters it belongs to. This leads to the definition of a location. A location can be either a single cell or a cell cluster. Important locations, where the user spends a large portion of her time, are called bases. The algorithm described in [8] solves the problem of learning personally important bases within the mobile phone device. Having this algorithm at hand we could simply log events captured when the user entering and leaving a base.

### 3.5 Event-Driven System Modeling

As explained earlier, the logging server described in this thesis is concerned with events triggered by user actions or events that in some way describe the environment where the user is, and thereby reflect the user activities. Sending and receiving SMS falls into the former category while capturing near-by Bluetooth devices falls into the latter category. Since the considered events are triggered by user activities at discrete points in time, the logging server observes the event occurrences discretely. We associate this behaviour of the logging server with the discrete event system described in [2].

Using the event modelling methodology described in [2] with several finite state automata running in parallel, we can handle different types of events originating from independent sources. Thus each event handler has been implemented as a finite state automaton.

A finite state automaton (FSM) is composed of a finite number of states, transitions between those states, and actions. A transition specifies how the automaton can proceed from one state to another, and is triggered by an associated signal. An action is invoked in response to a transition. In Figure 3.3 and 3.4, we illustrate automata graphically as state transition graphs.
Thus, states are represented as nodes, transitions are represented as directed edges between two states, and actions are performed at state changes. When the logging server is initialized, each automaton is in its **Idle** state. A transition to the **NewEvent** state (see Figure 3.3) is enabled in an automaton, when an associated event occurs and the occurrence is detected by the automaton. After one or more actions have been performed to handle the captured event, the automaton transits to the **Report** state. Lastly, when the event is reported for storage, the automaton transits back to the **Idle** state waiting for new event occurrences. This general approach will work for most events. However, for capturing Bluetooth events a slightly different automaton is needed.

**Figure 3.3 The general FSM for handling events**

Bluetooth device discovery is triggered by the logging server itself at regular intervals. When a discovering process starts and the first Bluetooth device has been detected, the automaton transits from the **Idle** state to the **ScanDevices** state (see Figure 3.4). The automaton will stay in this state, until no more Bluetooth devices are discovered. Then the automaton transits to **ScanCompleted**, and afterwards to **Report** and **Idle** states.

The parallel running finite state automata design ensures more efficient usage of resources e.g. CPU, by the means of asynchronous event handling\(^1\). We will see in next section how this can be achieved on the Symbian platform.

---

\(^1\)Asynchronous event handling indicates that an event is handled independently of other events, thus different event handling processes can be running in parallel.
3.6 Event Handling System

The system design of Symbian OS is optimized for efficient event handling. All Symbian programming is event-based, and the CPU can be switched off when applications are not directly dealing with an event [6]. Correct use of the event-handling system helps to ensure efficient software. To understand the Symbian event model one must understand its most important building block: the active object.

3.6.1 The Principle of Active Object

Active objects are the fundamental part of the event-handling mechanism in the Symbian OS. Each Symbian thread has a single active scheduler, which cooperates with one or several active objects to handle events asynchronously. Active objects provide non-preemptive multitasking (see Section 2.2.3) within the context of a single thread. The benefit of using active objects is that active objects do not have to switch context, do not require the same kind of sharing disciplines as are needed in multithreading systems. In addition, the active objects reside in their own thread of control, and therefore provide a high level of system integrity.

How an event-handling thread with an active scheduler and active objects works can be seen in Figure 3.5.

![Figure 3.5 Active object working process](image)

Each event-handling thread may have many active objects, but it contains only a single active scheduler that keeps track of the active objects. Each active object is associated with one object called asynchronous service provider in the sense that it completes the requests issued by the active objects asynchronously. When an active object makes a request to a
service provider, it calls `SetActive()` to set a flag on itself indicating that this active object has issued a request, see (1) and (2) in Figure 3.5. The service provider completes the request asynchronously (3) which results in an event (4). This means a signal is sent to the active scheduler, and a completion code is posted into the `iStatus` variable of the requesting active object (5) indicating the issued request is completed. When the active scheduler detects an event (6), it scans through all its active objects, searching for one that has issued a request and for which the request completion code has been set. Then the active scheduler calls the `RunL()` function of the responsible active object (7) to perform a certain task (8). At the same time the active scheduler resumes waiting for new events.

Using active objects achieves concurrency within a single thread and enhances software efficiency. The finite state automata described in Section 3.5 can be implemented in respective active objects running in parallel in the same thread. In each automaton, a transition from one state to another is enabled by issuing an asynchronous request, and actions performed at each state change are handled in the `RunL()` function. An example code can be seen as following:

```cpp
void CPhotoEventHandler::StartL()
{
    Cancel();       // Cancel any request, just to be sure
    iFsSession.NotifyChange(ENotifyFile, iStatus, iImageFolder);
    iState = EPhotoEvent;
    SetActive();
}

void CPhotoEventHandler::RunL()
{
    switch(iState)
    {
    case EIdle:
        break;
    case EPhotoEvent:
    {
        if (IsPhotoEvent())
        {
            HandlePhotoEventL();
            iState = EReport;
            TRequestStatus * status = &iStatus;
            User::RequestComplete(status, KErrNone);
            SetActive();
        }
        break;
    case EReport:
    {
        ReportEventsL();
        StartL();
    }
    default:
        break;
    }
}
```

This piece of code implements the automaton which handles the photo events and is embedded in the `CPhotoEventHandler` active object. The `CPhotoEventHandler` connects itself to the file system using a file system session (`iFsSession`) and is at the `Idle` state when
initialized. The \texttt{StartL()} function includes actually the actions performed at the \textit{Idle} state. Through the \texttt{iFsSession}, the \texttt{StartL()} function requests a notification of file changes in the specified \texttt{iImageFolder} which points out the folder where new photos are created by the camera. The \texttt{StartL()} function also sets the next state to transit to the \texttt{EPhotoEvent} and calls the \texttt{SetActive()} function. When a new photo is created in the \texttt{iImageFolder}, the \texttt{iFsSession} sets a completion code to the \texttt{iStatus} variable in the \texttt{CPhotoEventHandler} active object, and notifies the underlying active scheduler about the completion. When the active scheduler detects this notification, it scans over associated active objects, and finds out the \texttt{CPhotoEventHandler} has issued a request and the request is completed. At that time the \texttt{RunL()} function in the \texttt{CPhotoEventHandler} is invoked by the active scheduler, and the automaton is transited to the \texttt{EPhotoEvent} state. When the photo event has been handled at this state, the next state to transit is set to the \texttt{EReport}. The \texttt{CPhotoEventHandler} then calls the \texttt{SetActive()} and generates a request completion event itself to allow the active scheduler call its \texttt{RunL()} function again in order to trigger the state transition. In this way the state is transited to the \texttt{EReport} state, and the photo event is then reported for storage. The \texttt{StartL()} function is invoked again waiting for new incoming photo events.

Furthermore, active objects are significant in the Client/Server architectures in Symbian, where a server is implemented as a single event-handling thread which handles events asynchronously using active objects.

### 3.6.2 Client/Server Framework

In Symbian, servers (see Section 2.2.2) are widely used to handle resources on behalf of multiple clients. These include the Windows Server, File Server, Telephony Server, and Messaging Server etc [6, 7]. A server can be one of three types: constantly running system servers, transient servers which are started by the first connection request and terminated when their last client is disconnected, and servers which are started and terminated along with the application [6].

Servers run within their own process or thread, and requests from clients are made across thread boundaries. There are two ways to cross the thread boundary: message passing and inter-thread data transfer (the server can access a client’s address space for transferring more data), both supported by the Symbian kernel [6]. Message passing is used in all Client/Server communications. When a large amount of data needs to be communicated between a client and a server, only pointers to the client-side address space are passed to the server as messages and the data can then be inter-thread read from, and written to, the spaces to which the pointers refer.

For implementing servers in Symbian, the provided Client/Server API is used. There are four main concepts in the API: server manager (\texttt{CServer}), server-side and client-side session (\texttt{CSession and RSessionBase}) and message (\texttt{RMessage}).
A server manager is an active object. It is responsible for establishing new sessions between clients and the server. When a client issues a request, the server manager handles the request asynchronously by channelling the request to the appropriate server-side session and lets the session perform the requested services. Then it listens to the next incoming request.

A session is the channel of communication between a client and a server. There are server-side sessions and client-side sessions. The client-side sessions are responsible for converting data into suitable message format and sending messages to the server, whereas the server-side sessions receive and handle the messages. Besides the server-side and client-side sessions, there are kernel-owned sessions linking the corresponding sessions.

The data structure passed between a client and a server is the message. For sending small amounts of data, the message contains the actual data. When large amounts of data are to be sent, the message contains pointers to the client-side address space which the server can use for reading data from and also writing data back to the client. Besides these data the message also contains a code specifying the client request type.

Assuming a large amount of data to be communicated between a client and a server, when the client issues a request to the server, see (1) in Figure 3.6, it arrives at the server side as a message. The message includes the operation code specifying the request type and an array of pointers pointing to the client-side address space. The request is first channeled to its appropriate server-side session by the server manager (2). The server-side session then examines the operation code (3) and a proper action is taken to perform the request (4). Using the pointers pointing to the client-side address space, the server-side session...
reads/writes data from/to the client (5) and finally signals completion of the client request (6).

### 3.6.3 The Client/Server Architecture in the Project

In this project we wanted to design and implement a logging server, which logs events that can be used by more applications than Affective Diary, that is, to design a general purpose logging API. The logging API can be used by third party applications, such as the Affective Diary application, to access logged events. To achieve this, we made use of the Symbian Client/Server framework described in Section 3.6.2, and developed a Client/Server architecture for the project, which is illustrated in Figure 3.7.

![Client/Server architecture](Figure 3.7 Client/Server architecture)

A client application issues a request using the logging API interface. The logging API is responsible for translating the client request into one or several requests that can be understood by the client-side session. In the client-side session these requests are formatted into suitable messages and sent to the server. At the server side, the messages are dispatched to an appropriate server-side session, which handles them by performing the corresponding services. Finally, an acknowledgement is sent back by the server-side session to inform of the completion of the request.

### 3.6.4 Communication Service

A problem we wanted to solve in this project was to automatically transfer information between the logging server on a mobile phone and a PC-based Affective Diary application. Thus, we designed a communication service running besides the logging server to handle this. Our approach was based on writing and reading commands to files.

The communication service is a client to the logging server. When a command file is created in a certain folder by a PC-based Affective Diary application, the communication service is notified. It parses the command file and the corresponding command is performed by issuing requests to the logging server. Two kinds of commands are supported by the communication service: get events and remove events. One can specify the time interval within which the get/remove events commands should take place. For instance, a command can be specified as: get events logged between the 02/02/2007 at 08:00 and the 02/02/2007 at 23:30. The logging server then obtains the requested data from the database and generates an XML file containing the data. A piece of the generated XML file can be formulated as
the following XML code.

```xml
<Event type="SMS">
  <Time>02-02-2007T14:00:06</Time>
  <Description>+46704463248</Description>
  <To>me</To>
  <From>+46704463248</From>
  <Subject>HELLO</Subject>
</Event>
...
<Event type="Call">
  <Time>02-02-2007T14:20:13</Time>
  <Description>No Reply</Description>
  <RemoteParty>0704463248</RemoteParty>
</Event>
...
<Event type="Photo">
  <Exist>True</Exist>
  <Path>C:\documents\media_files\image\unfiled\711E0011.jpg</Path>
  <Time>02-02-2007T15:00:14</Time>
</Event>
...
<Event type="BT">
  <Time>02-02-2007T15:47:00</Time>
  <Description>0e798f2e0</Description>
  <Id>0e798f2e0</Id>
  <Name>MobiTip5</Name>
  <Disappeared>False</Disappeared>
</Event>
```

Each event is described by a set of tags in the XML file. For instance, a photo event is described by the time indicating when the photo was taken, the file path to the photo, and a flag telling if the photo still exists; and a phone call event is described by the occurrence time, the remote party telephone number, and a phone call status description indicating the direction of the call and if it was established.

Finally a result file containing the full file path to the XML file is returned to the PC-based Affective Diary application. The application can then transfer the entire XML file and also retrieve the relevant data, such as a photo file, from the mobile phone. These materials are later used for representing the logged events in the Affective Diary application. In the case of removing events, the result file will only contain a code indicating whether the process has succeeded.
3.7 System Architecture of the Logging Server

After having seen the different components of the logging server, we introduce the system architecture of the logging server.

![Diagram of the logging server architecture](image)

As illustrated in Figure 3.8, GSM services, Bluetooth services and phone resources on a mobile phone communicate with user activities and generate events that we want to log. These events can be accessed by Symbian public APIs and third party software. Each of these events is handled by an appropriate event handler and reported to a database for storage. Through a general logging API, the logged events can be accessed by client applications such as the communication service, which acts as a middleware, and performs the communication requests from the PC-based Affective Diary application. The events, event handlers, database and the logging API compose the logging server.

3.8 Design Patterns

Design patterns are recurring solutions to commonly occurring problems in software design [5]. They describe how to solve problems and can be used in many different situations. In this project we wanted to create a logging system that is easily extendible in terms of adding new event handlers by third party developers. To do this it is very natural to use design patterns as they provide solutions to such common problems in software design.
Generally, design patterns can be classified into three main categories [5]: Creational patterns dealing with object creation mechanisms, Structural patterns representing relationships between entities, and Behavioural patterns providing solutions for interactions between entities. We begin with selecting appropriate design patterns to implement the different components of the system.

The Factory method and Abstract factory creational patterns are often used together to abstract away the object instantiation process. They define an interface for creating objects, and hide how the objects are created from the overall system. The Factory method pattern is used when a class cannot anticipate the type of objects it must create and relies on its subclasses to specify the object instantiation, whereas the Abstract factory pattern provides a way to encapsulate a set of individual factory methods that have a common theme [5]. The factory patterns make the overall system independent of how its objects are created and composed. The Figure 3.9 illustrates an example of how these factory patterns were used in this project.

Handling a type of events requires implementing a corresponding event handler. The logging system is extendible in the sense that it provides an easy way for other developers to add new event types and event handlers. It is also necessary for the logging system to be easily extended to include new sets of event handlers when porting to other platforms, since the implementation of event handlers depends on the hardware and software environments. These are achieved by using factory patterns. The class diagram in Figure 3.9 illustrates that the CEventHandlerFactory is an AbstractFactory class and defines a factory method for creating event handlers, whereas a derived class, CDefaultEventHandlerFactory, specifies which event handlers to create by defining a set of individual factory methods. A new event handler can be easily attached to the structure by defining a new factory method, while a new set of event handlers can be included by implementing a new class derived from the AbstractFactory class.
Among structural patterns, the \textit{Facade} pattern helps provide a simplified interface to the overall system, which makes the system easier to use and understand. In this project the \textit{Facade} pattern was applied to the \texttt{CLoggingServer} class as illustrated in Figure 3.10.

The \texttt{CLoggingServer} class interacts with the rest of the system, e.g. \texttt{CEventDatabase} class and \texttt{CEventHandler} classes, and provides a common interface for different \textit{Clients} to access resources in the system. This helps reduce dependencies of outside code on the inner system, thus allowing more flexibility in developing the inner system.

The \textit{Observer} pattern is a behavioural pattern, which is used when one or more objects (called observers or listeners) are registered to observe events which may be raised by observed objects. The class diagram in Figure 3.11 illustrates how this pattern was used in this project.

An event observer is registered in each of the event handler classes. When an event occurs, the event observer observes the occurrence of the event and reports it to the \texttt{CEventCache} which in turn reports the event to the \texttt{CEventDatabase} for storage.
3.9 Problems and Solutions

There were several problems we met during the implementation. We describe each of the problems as well as the decisions made and respective solutions to deal with them.

Bluetooth Discovery

When implementing Bluetooth device discovery, some difficulties in finding an appropriate time interval to trigger the discovery were met. As the process of Bluetooth discovery was designed to be triggered by the logging server periodically; how long should the interval between each discovery be to ensure that the logging server still runs efficiently and does not miss information? A strategy allowing the time interval to be chosen automatically by the logging server itself was selected: A default short time period of 4 minutes was used for initializing the time interval. During a discovery process, if changes are seen between the previous discovery and the current process, i.e. new Bluetooth devices have been discovered or devices that were discovered in the last discovery have disappeared, the time interval to next discovery is set to the default value of 4 minutes. The reason of using a short time interval between discoveries that have changes is that if a change happens in a discovery, some or even more changes might happen in the following discovery, since the user might at that time be in a dynamic environment, such as in a crowd or travelling. If no changes have been seen in the previous three Bluetooth device discoveries, the time interval was increased by 2 minutes. The interval is set back to 4 minutes as soon as a change has been detected or when a maximum value of 20 minutes is reached.

Filtering of Bluetooth Devices

Another problem met during the implementation of Bluetooth device discovery was how to implement the filtering solution we described in Section 3.4.5, i.e. how to log only the events describing when a Bluetooth device first appears and when the device disappears? To solve this problem, two lists were used in order to keep information of discovered Bluetooth devices and existences of them. Let the logging server produce an Event object for keeping information of each discovered device. The first list is used for temporary storage of the Event objects produced during a discovery process. The second list has exactly the same length as the first list and uses either a 1 or a 0 as the entry value. We will see how these lists are used in the filtering process:

![Figure 4.1 An example of the two lists before a Bluetooth device discovery process](image)

Assuming as the result of a Bluetooth device discovery a number of Bluetooth Event objects are temporarily stored in the first list. As an example illustrated in Figure 4.1, the first list contains currently BT 1, BT 2, BT 3 and BT 4, which indicate different Event objects. For
each Event object in the first list there is a corresponding value of either 0 or 1 in the second list. If the second list consists only of 0:s, it indicates that the first list is in its originating state, i.e. ready for next Bluetooth discovery. During a discovery process, Bluetooth devices are discovered continuously. For each discovered device the first list is searched through to check whether the device is already present. If it is not the case, an Event object is created describing the discovered Bluetooth device and appended to the first list for temporary storage. An entry with value 1 is appended to the second list as well. At the same time this Bluetooth event is logged into the database for permanent storage, as it describes the first appearance of the device. If the same Bluetooth device is found in the first list, the corresponding entry value in the second list is changed to 1, indicating that the device has been seen again. Since only events that describe the first appearance and the disappearance of a Bluetooth device are logged, the occurrence of this device should be ignored. As shown in Figure 4.1, when BT 1, BT 3, BT 5 and BT 6 are discovered during the next discovery, the following changes are done in the two lists: in the second list, the entry values which relate to BT 1 and BT 3 are changed to 1, in order to describe BT 5 and BT 6, two Event objects are created and appended to the first list, and two entries with value 1 are then appended to the second list. The two lists are then shown in Figure 4.2.

![Figure 4.2 An example of the two lists after a Bluetooth device discovery process](image)

After the discovery process, these two lists must to be checked once more. This time entries with value 0 in the second list are sought after. Since value 1 has been set to indicate occurrences of corresponding Bluetooth devices during the discovery process, the 0:s left in the list point out which devices have disappeared. The disappeared Bluetooth Event objects are removed from the first list and these events are logged with disappearance time to the database. At the same time, the 0:s are also removed from the second list, and each entry containing value 1 is now initialized to 0.

![Figure 4.3 An example of the resulting lists](image)

The resulting lists from these two steps are shown in Figure 4.3, where the first list holds the Bluetooth Event objects indicating devices discovered in the recent discovery, and the second list consists of 0:s. At this point, the lists are ready to be used for the next Bluetooth device discovery.
Other decisions made in Bluetooth discovery
In order to discover near-by Bluetooth devices the Bluetooth service on the phone must have been turned on. No Bluetooth device discovery is triggered if the Bluetooth service has been turned off. In the situation when the Bluetooth service is turned off by the user after a number of discoveries, all discovered devices from the most recent discovery will be logged with a description indicating that the Bluetooth service has been turned off. When the Bluetooth service is turned on again, the logging server will trigger device discoveries as usual.

Besides the Bluetooth device name, address and existence, it was also interesting to log the information of offered services on each discovered Bluetooth device. But on the other hand, the Bluetooth service discovery requires much more resources of both CPU and memory, considering that the service discovery will be performed on each discovered device in each device discovery and a device may offer a number of services. It may also slow down the speed of the device discovery. Thus, we decided to leave the Bluetooth service discovery implementation to future work.

GSM Cell ID
We had an algorithm [8] at hand that makes use of the GSM cell IDs and handles the problems of learning the personally important places such as office or home. It would be interesting if we could translate the cell IDs to real location names. We found a global location based information service called CellSpotting (www.cellspotting.com), which works on Symbian based mobile phones and is supported by distinct operators in different countries. Using this service a cell ID number can be easily converted into the name of a geographical location. Unfortunately, the current version of CellSpotting works only on Nokia Series 60 and Siemens SX1 Smartphones. Having SonyEricsson P910 as target phone, we could not make use of this service. Therefore the GSM cell ID information events were left to handle in future work.

MMS
As mentioned in Section 3.2.1, P910 has its own version of MMS API [12] to access MMS messages. This API is actually based on the Nokia Series 60 MMS API, and is therefore not compatible with the P800 and P900 phones. Since the current version of the PC-based Affective Diary application did not include the MMS events, and we wanted the logging server to work on P900 as well, we left the implementation of the MMS event logging to future work.

Controlling the size of the Database
To ensure that the database which stores the logged events would not be expanded infinitely in size, we decided to use an auto-controlling strategy to control the size of the database and remove a number of rows from the database automatically when needed. We implemented the strategy as follows: the logging server activates a control of the database size automatically every midnight. In the case that the free memory on the phone is below 50 percent of the entire user memory and the size of the database is above 50 percent of the
used memory, which indicates that the database is actually large in size, we will free a piece of memory allocated by the database by removing a number of rows containing the oldest events in the logging history.

Configurability
A goal to achieve in this project was to make the logging server configurable by client applications. This was done through the general logging API. Available configuration functions handle the following issues:

- The default time interval value of 4 minutes between Bluetooth device discoveries can be configured.
- In order to restrict the running time of the discovery process, a timer has been set to stop the discovery process when it processes in a longer time. This timer can be configured.
- If desired, the filtering of Bluetooth devices can also be disabled, which means each discovered device is logged repeatedly.
- The percentages used when controlling database size can also be configured.

Start on boot
The logging server should be started automatically each time when a mobile phone reboots. We used the concept of Recognizer in Symbian [13] in order to achieve this. A recognizer is basically a plug-in pre-compiled library that is loaded by the Application Architecture server¹, and handles some kinds of documents or files by examining data in it in order to recognize a certain data type. We were not really interested in handling files, but a specific property of the recognizers is that they are loaded at start up. We implemented a recognizer and embedded code in it that could start the logging server. In this way, the recognizer is invoked at mobile phone restart and the logging server is then started automatically.

General Logging API
As a client application to the logging server the developed communication service used only a few of the functions offered by the logging API. We implemented a simple test program as another client application in order to verify all other available functions, such as configuration functions and functions developed to return a list of Event objects describing the requested events. What the test program did was simply to issue requests using each of the functions offered by the logging API. By running this test program on the phone, we could verify whether the logging server behaved as we configured and whether the returned data were correct.

¹Governs file formats and application launching in Symbian.
4 Verification and Results

There were several goals to achieve by the logging server: stability, reliability and data integrity, i.e. the logging server should work stably in the background, not disturb the users’ daily use of their mobile phones and not crash. The logging server should also work correctly, i.e. no data should be distorted or lost. Furthermore, data should be identically maintained during any operation, such as storage, retrieval and transfer. To verify the functionalities of the logging server we performed several verification tests.

4.1 Stability

Setup
Two test users performed a verification test by running the logging server on two P910s for a period of at least 30 days. The users observed whether the logging server starts automatically at mobile phone reboots and whether the battery life is significantly affected by running the logging server. The users tried also to “use” the logging server as much as possible during the verification, i.e. the users tried to “generate” events that are handled by the logging server by sending SMS text messages, making phone calls and taking photos. Furthermore the user tried also to spend time in different environments such as in a crowded shopping centre or at work where the environment is much more stable in order to verify the Bluetooth device discovery functionalities. Whether the events are logged at their occurrence time can be verified by checking in a debug text file, which holds text print-outs from the logging server, on the phone. Any disturbances or system crashes caused by the logging server were reported by the users.

Result
The logging server has been experienced as stable software by the users. It started each time the users restarted their mobile phones. The logging server did not disturb the users’ normal use of their phones. The battery life was not significantly affected by running the logging server. All events generated by the users’ activities were logged when looking in the debug text file. However the logging server was not completely stable in the beginning of the verification test because of some implementation mistakes. After having been improved, the logging server ran stably in the later phase of the verification test, and no crashes were experienced by the users.
4.2 Reliability

Setup
A detailed consistency verification test was carried out by one of the test users. In the course of two days, the user verified the logging server by performing a given predefined set of tasks. Each of these tasks should generate a certain event. These pre-known events were compared with the resulting event logs to verify the consistency of the logging server. The following tasks were included:

- send a SMS message
- receive a SMS message
- dial a phone number but get no reply by the remote party
- get a phone call but do not answer
- dial a phone number again and get reply by the remote party
- get a phone call and answer it
- take a new photo
- remove the taken photo
- enable a near-by Bluetooth device
- enable another near-by Bluetooth device
- disable the Bluetooth service on the phone
- enable the Bluetooth service again
- turn off the transmitting functions (turn on flight mode)
- turn on the transmitting functions again (turn off flight mode)
- disable one of the near-by Bluetooth devices
- disable the other Bluetooth device

The tasks were performed at equal time intervals of 10 minutes in a stable environment. In this test we verified the incoming/outgoing SMS and phone call events logging, photo taking/removing events logging, and Bluetooth device discovery events logging. For verifying the Bluetooth device discovery we enabled and disabled one Bluetooth device at a time to simulate the behaviour of Bluetooth devices. We also tried to turn on/off the Bluetooth service on the phone as well as go in flight mode to ensure that the logging server handles such situations and logs events when the Bluetooth service turns off.

Result
By requesting the logging server to generate an XML file (containing logged events) covering the period when the test was performed, the user found that the pre-defined events were logged correctly at the time stamps they should occur and contained exactly the information we defined.
4.3 Data Integrity

Setup
Data integrity of transfer between the logging server on the phone and a PC-based Affective Diary application was also verified. This test was also performed by one of the test users. The PC application developed at SICS can automatically request logged events from the logging server when a connection between the phone and a PC is established. The events requested are from the point in time when the last synchronization was done by the two devices to the point in time when they are connected again. The user connected his phone to the PC application once a day and compared what was represented by the Affective Diary application on the PC and which events were actually logged on the phone (can be seen in a generated XML file). The comparison aimed to ensure that the events were identically maintained on both sides, i.e. no data were lost during the transfer.

Result
The user experienced that the logging server worked correctly during the transfer and the data were identical on both sides. The logging server even succeeded in transferring large amounts of data, e.g. event logs over several weeks.
5 Discussion and Conclusions

A general discussion regarding the developed logging server is held in this chapter, e.g. the difficulties we found and the shortcomings of the software. We present the important results of this project and also future improvements and usages.

5.1 Discussion

As we have seen by the verification tests, the logging server worked stably and correctly, and no data were lost during any transfer. As we have described in Chapter 3, the logging server could be used in different application scenarios, and was designed to be easily reused and extended. However, the program can still be improved.

The logged Bluetooth events could not completely represent near-by Bluetooth presence. Since Bluetooth devices can come in and go out of range of other devices easily and quickly, and the Bluetooth device discovery is triggered periodically by the logging server, it may happen that some of the devices that have been near-by the logging server in a shorter period can not be detected. Since the discovery process can even be stopped by a timer when it takes a longer time in execution, it may be possible that some devices considered as having disappeared might still stay in the range. Furthermore, since the frequency of Bluetooth device discovery slows down when the environment has not been changed for a longer time, new Bluetooth events can only be detected when the next discovery is triggered. At that time some of them may have already been disappeared. Thus, we may miss logging some Bluetooth events, but on the other hand, as long as the Bluetooth device discovery is triggered periodically, we have to get along with these problems.

During the verification tests, we also found that it was complicated to verify the Bluetooth event logging. Since we performed the verification test in a relatively stable environment with pre-defined Bluetooth events (Bluetooth devices with arranged appearances and disappearances), we could easily see that the logging server logged these devices correctly. But in an uncontrolled environment, with many Bluetooth events occurring without our knowledge, it was difficult to verify whether all logged Bluetooth events happened. However we experienced that the logging server logged large numbers of Bluetooth events in a dynamic environment, whereas few Bluetooth events were logged when the user spent her time in a more stable environment.

Using the parallel running finite state automata design and implementing them as active objects in a single thread, we ensure more efficient usage of resources e.g. CPU, and thereby
ensure a more efficient software. However the logging server can still be optimized. When implementing the filtering solution in the Bluetooth device discovery (see Section 3.9), we chose to use one list to keep temporally generated Event objects and another list to keep information of the existence of each of them using 0:s and 1:s. In fact, we found that it could be more effective to replace the second list with an unsigned 32-bits integer in which each bit represents the existence information. In the case when the number of discovered devices exceeds 32, which is seldom seen, a list of integers could then be built to hold the extra information.

We used design patterns (see Section 3.8) to ensure the reusability and extensibility of the logging system. However this could not be verified by us, since the verification of how easily it is to reuse the code and extend the logging system should be performed by a third party developer.

A study of the software for a prolonged period of several months was being performed on P900/P910 mobile phones by five persons, but the experiences and results from that study could not be included in this thesis because of the testing time. However, no problems with using the logging server have been reported.

5.2 Conclusions

As a result from this Master’s project work, the developed logging service meets the goals set in the beginning rather well. The software logs events relevant to the Affective Diary, filters data to avoid logging redundant data, and handles data transfer between the software and a PC-based Affective Diary application. It runs automatically and in the background, and does not disrupt a user’s normal use of the mobile phone. It works stably, correctly and reliably, and data are identically maintained during storage, retrieval and transfer. Furthermore, the logging service provides a general logging API and allows different client applications to make use of the offered services. Also as desired, this service is designed to be easily reused and extended.

5.3 Future Work

The reusability and extensibility of the logging system ensure that the program can be easily developed by third party developers in future work, e.g. new events such as MMS and GSM cell ID information events can be handled by extending the system. The program code can also be optimized to execute even more efficiently. The services offered by the logging service can be used by different client applications: there will be another Master’s project under the Affective Diary project at SICS which makes use of the logging service and focuses on developing a GUI in order to offer a user-friendly way to review and reflect on the logged events on the mobile phones. For transferring data between the logging server on
a mobile phone and a PC application we chose using the file system to exchange information, while other methods such as Bluetooth or infrared connectivity between the devices can be used in future work.
References


