Free File Transfer

Using SMS & Symbian C++

E 05

Participants:
Jesper Mouritzen
Anders Bo Pedersen

Supervisors:
Søren Debois
Kåre J. Kristoffersen

Preface

This project deals with the design and implementation of a data transfer program that uses the Short Messaging Service to transfer data between mobile phones. The program has been developed using Symbian C++. The program is designed to run on the Nokia Series 60 phones. The project has proved that it is possible to transfer files using the SMS protocol. Problems regarding the handling of SMS messages by wireless operators, introduce problems that prevents the project prototype from working in all situations. The results of this project could be elaborated into a working product and the concept of SMS data transfer has therefore been proved.

The attached CD contains the following:

- The project report in DOC and PDF formats under the folder 'Report'.
- The project source code as a Code Warrior v.3.1 project under the 'Code' folder.
- A Series 60 program build ready to be installed on a S60 Smartphone under 'SIS'.

In the case of a malfunctioning CD, the CD contents can be downloaded from:

http://www.itu.dk/people/abp/SDT_CD_backup.zip

____.

Jesper Mouritzen

Student nr. 2130790694

____.

Anders Bo Pedersen

Student nr. 1818610966

1 Introduction 5
2 Project Statement 6
2.1 The Idea 6
2.2 Problems 6
2.3 Limitations 7
2.4 Method 8
3 Technical Analysis 9
3.1 Why Symbian? 9
3.1.1 BREW 9
3.1.2 Windows CE & Windows Mobile 10
3.1.3 J2ME 10
3.1.4 Symbian 11
3.2 Symbian C++ features 11
3.2.1 Event driven design 11
3.2.2 Asynchronous services 12
3.2.3 Active Objects 12
3.2.4 User Interface 13
3.3 Definition of the SMS 13
3.3.1 The Basics 13
3.3.2 SMS Transfer Process 14
3.3.3 Expected bandwidth 15
3.4 Ensuring Quality of Service 16
3.4.1 Reliability of the SMS service 16
3.4.2 Validating files 17
3.5 Encoding before Transmission 17
3.5.1 Compressing data 17
3.5.2 Encrypting data 18
3.5.3 Preserving Battery Time 18
4 Program design 19
4.1 Choosing a file 19
4.2 Dis-assembling a file 19
4.3 Sending a message 20
4.4 Waiting for and receiving a message 20
4.5 Re-Assembling a file 21
4.6 Timing the process 21
4.7 UML class Diagram 22
5 Implementation 23

5.1 Symbian; Getting acquainted....23
5.1.1 Naming Conventions 23
5.1.2 Where is the console? 23
5.1.3 The time waist of emulation.... 24
5.2 Class descriptions....24
5.2.1 The foundation.... 25
5.2.2 Timing.... 26
5.2.3 Views 27
5.2.4 File handling 28
5.2.5 Messaging 30
5.3 Experiments and Discoveries 35
5.3.1 LeaveScan.... 35
5.3.2 Socket communication 35
5.3.3 The Nokia Connectivity Framework.... 36
5.3.4 Porting the Program 36
5.3.5 Low Memory Tool 37
5.3.6 Full Outbox Crash.... 38
5.4 The Final Prototype 38
5.4.1 The first OTA test 39
6 Benchmarks 40
6.1 Testing on mobile phones....40
6.1.1 Processing messages at sender 41
6.1.2 Processing messages at the receiver.... 42
6.1.3 From sender to receiver 43
6.1.4 Total bandwidth 44
6.2 Testing on the emulator 44
7 Conclusion 47
7.1 The Symbian Experience....47
7.2 The Product 48
7.3 Reached goals....48
8 References....51
9 Appendix I Project code 53
10 Appendix II Test Results....54

1 Introduction

The growth of the mobile device industry has exploded over the last few years. New technologies are popping up every day and mobile phone devices are becoming more and more advanced. This growth has of course attracted investors from all over the world and has resulted in many new mobile phone operators being established. In their effort to attract new customers many of these mobile phone operators are currently offering many subscription types. These include services that are not presently used to their full extent. Services like ‘free SMS’ and ‘cheap GPRS’ could in this groups opinion potentially be used/exploited as cheap, or even free, data transfer services given the proper software.

Almost all operators in Denmark have their own variety of the ‘free SMS’ subscription type at writing hour. This makes a program that uses such services for free data transfer, a very useable application for a large target audience. Imagine the possibility of sending or sharing ring tones, music or even video via SMS for free! Any teenager today would find that possibility appealing. Investigating the option of sending data via a ‘free SMS’ subscription, is therefore obvious.

Since GPRS is expected to become more or less free in a near future with the introduction of ‘flat rate’, developing a data transfer program that supports the technology could prove to result in acclaimed application. Furthermore, Danish operators are selling mobile phones with preinstalled GPRS settings, which presumably makes the introduction of the technology easier, especially among the younger generation who replaces their mobile phones frequently. So GPRS is also a technology that could or should be examined in the quest of achieving cheap data transfer on mobile phones.

2 Project Statement

2.1 The Idea

The purpose and idea of this project is to explore whether SMS and GPRS are protocols that are suited for wireless file transfer. We will examine whether SMS and GPRS data transfer can maintain expected service standards such as bandwidth and reliability. We will also map the possibilities within this field, examine the limitations of the protocols in the context of data transfer and come up with solutions that can settle these limitations. Finally we will also explore whether the low level memory access of Symbian enables developers to challenge the hardware restrictions of mobile devices that e.g. J2ME developers simply have to accept.

The group will in this project strive to implement a working prototype that uses at least one of the above data transfer technologies to transfer data at as low a cost as possible. This prototype should be seen as a 'proof of concept' and could possibly serve as a basis for a later more elaborate Symbian project that incorporates better GUI design and seamless integration of the data transfer program with common file browsing in Symbian. Even more complex projects that investigate the possibility of peer to peer networks and file sharing based on the outcome of this project, could also be a possibility.

If the results of the projects reveals high bandwidths, our project could support existing projects for wireless devices that require such. Such projects could be the p2p project Symella [23] that supports the popular Gnutella sharing protocol or the wireless version of Skype [24] soon to come.

2.2 Problems

The problems of the project are:

- Is it possible to transfer files from one phone to another for free or at low cost using the SMS or GPRS protocol?
    - Are the SMS and GPRS protocols reliable and what bandwidths are obtainable?
    - How should a data file be segmented into chunks to accommodate either of the listed protocols before transmission?
    - How big a file can be transferred between 2 phones?

- Is it possible to stream data between 2 phones?
- How does one insure that all chunks of a segmented file have been received at the destination phone and how should possible lost packages be retransmitted?
- How should a file be reconstructed from packages at the receiving phone and how should such a file be validated for consistency?
- Is package encryption an issue that should be regarded and if so, how?

2.3 Limitations

To give an impression of the conditions under which this project will be developed, this section lists the limitations under which we will work.

- The software developed for this project will be made solely for the Nokia Series 60 line of phones. We feel that it is necessary to mention this since 'porting' is a big issue in mobile phone development. Throughout the project the testing models will be The Nokia 6600 and Nokia 7210.

- The group will use MetroWorks CodeWarrior v.3.1 Personal edition as our IDE. We will use the Nokia Series 60 SDK DP v2 FP3 which is found at www.newlc.com and resembles the Symbian OS v.8.1a. We have chosen the above SDK since it has libraries that we might find time to explore. We do realise though that the above testing models do not support all features of the SDK.

- For inter-phone transmissions the group will use the Nokia Connectivity Framework v.1.2 which should enable us to emulated data transmission between phones on a PC.

- To start with, we will explore the SMS protocol and therefore also limit the implementation to comprise this protocol only. If time allows it we will also explore and implement a solution based upon the GPRS protocol. This decision is based on the fact that a working program based on SMS and a 'free SMS' subscription would be a totally free service whereas a free GPRS subscription is less accessible with the Danish operators.

2.4 Method

No one in the group has prior experience with Symbian C++; hence this project is going to be an explorative one. Our implementation will hence be based on desktop research, supervisor interviews and code hacking.

The group hopes that its prior experience with software development for mobile devices using J2ME will ease the process of learning Symbian C++ since some terminology is expected to be shared amongst the languages.

The process of this project will rather traditionally be divided into the following phases as will the report.

- **Technical Analysis:** We will analyse the SMS protocol to find its weaknesses and strengths. This knowledge will be used to avoid pitfalls in the design and the implementation and used to solve the problems of the project. The expected bandwidth of our data transfer will be settled based on theoretical specifications.
- **Program Design:** Based on the problems and the analysis our initial program design will be done in this chapter. An initial class structure will be settled.
- **Implementation:** Problems, annoyances and achievement will be highlighted and elaborated upon and code candy will be exhibited.
- **Benchmarking:** The produced software will be tested and compared to benchmarks settled in the analysis. Bandwidth, stability and reliability are preliminary expected to be the main issues this chapter.

3 Technical Analysis

3.1 Why Symbian?

Many technologies for mobile device development are currently on the market. Some of the major technologies that have gained developer approval and user acclamation on both phones and PDAs through the last few years include:

- Binary Runtime Environment for Wireless (BREW)
- Windows CE & Windows Mobile
- Java 2 Platform, Micro Edition (J2ME)
- Symbian OS

When doing a project such as this one, technology research is a natural part of the development phase and should be used define to define the technology best suited for the task at hand. This chapter will do so.

3.1.1 BREW

BREW can be seen as an American initiative to provide developers with a platform that gives much of the low level functionality that Symbian also provides. BREW is coded in C and C++ which means that the language should enable developer to create advanced and fast applications. BREW has a strict authentication process when it comes to publishing applications. This process ensures that BREW applications work on a wide range of handsets once published.

BREW has from the beginning been very focused on making it easy for developers to publish their products and for users to obtain them. BREW supported handset have direct access to a wireless marketplace for games and apps that are accessible for purchase and download. The user therefore decides when and why to attain new software and can do this directly from the phone.

BREW is not that popular in Europe. The main reason for this is simply that most operators and phones here do not support the technology. This is also the reason why this group has not dwelt deeper into the technology and why we will not use it in this project.

BREW Main Website: http://brew.qualcomm.com
BREW online demo (very nice): http://brew.qualcomm.com/brew/en/operator/demos/demos.html

3.1.2 Windows CE & Windows Mobile

Obviously Windows CE & Windows Mobile are versions of Microsofts popular OS but for small computers and embedded systems. The main area of application for these two OS have from the start been Pocket PC and PDAs. Because both OS demand more memory and speed than supplied by the average phone, only very few Smartphone use either at writing hour. We have therefore decided not to investigate this technology further.

We would like to add though, that we see a bright future for the Windows Mobile OS because it is based on the popular C# and .NET framework. This should attract many developers in the time to come and by so exciting features.

Windows CE main page: http://msdn.microsoft.com/embedded/windowsce/default.aspx
Windows Mobile main page: http://www.microsoft.com/windowsmobile/default.mspx

3.1.3 J2ME

The probably most operator and handset supported technology in Europe is J2ME. The language has gained immense approval amongst both developers and users the last few years and is supported by almost all Smartphones from European manufacturers.

J2ME is obviously based on Java. Some say that Java is faster to learn and develop in than e.g. C++ which is one of the main advantages of J2ME. Besides being fast to develop in, J2ME shares many features supported by its older sibling J2SE. This has attracted many classical Java developers to the wireless segment and keep attracting *newbies* since the language is more human comprehensible and easy to learn. This is one of the main reasons for the technology's popularity.

So why not use J2ME for this project? Neither Java nor J2ME offers low level memory management which is a restriction in itself. Because J2ME run 'on top' a phone's OS the

technology runs under certain restrictions. This means that one can do almost anything with J2ME but at a restricted level. This project calls for the ability to send SMS messages. J2ME can easily do that. But because the developers of J2ME does not want to encourage misuse or abuse of user phones, restrictions of this feature had to be employed. Therefore the users must acknowledge sending a SMS each time this event occurs if implemented through J2ME as explained on p.34 of [6]. In practice this means pressing an 'OK' button per SMS. Since this project involves transparent sending of hundreds, maybe thousands, of SMS', this security restriction means that the group has chosen to disregard J2ME.

J2ME Main page: http://java.sun.com/j2me/index.jsp

3.1.4 Symbian

The Symbian OS offers the group all the features that the above discussed technologies do not. It gives low level memory access due to its Symbian C++ interface, it does not pose usability threats due to security restriction and it is supported on many phones currently used and shipped in Europe.

The only thing talking against using Symbian is the unstable future of the technology. The technology is not supported by all vendors and competing technologies are gaining terrain every day. The group has chosen to disregard this fact since nearly all Nokia phones uses the Symbian OS and since Nokia remain the biggest worldwide seller [10]. The group will use Symbian in this project.

3.2 Symbian C++ features

In the following chapter we will make a short description of some of the key issues in Symbian C++. Since Symbian C++ differentiates itself from traditional C++ in many ways, we will focus on the issues that have been important to this specific implementation and its specific functionality. This means that we will not spend time on Symbian basics such as two-phase construction and the cleanup stack even though they are not trivial.

3.2.1 Event driven design

Event driven design is a software design pattern that is build around services, requests and events. A service is a part of a program that can receive a request to carry out an order from a function some

where else in the program; the service is characterized by its ability to decide when the request is to be carried out as an actual event and for the requesting functions ability to ignore exactly when this happens. Services that control when to carry out the events they are presented and don't expect the caller to wait for an answer is called an asynchronous services.

3.2.2 Asynchronous services

Asynchronous services are used to separate the tasks of individual services so they can operate as atomic units. The trick is that only one service gets to carry out an event at any given time and that any event is promised not to be interrupted by any other tasks when first started. This concept is called *non-preemptive multitasking* and is one of the key issues when programming multiple threaded applications in Symbian C++, as the next chapter about *Active Objects* will discuss.

3.2.3 Active Objects

When programming software it is often necessary to make use of multiple threads to accomplish different tasks. In the case of this project, the tasks could include sending and receiving files as well as de-assembling and assembling files. In the Symbian OS it is possible, but not recommended, to implement multiple threading in the standard C++ manner. Instead it is suggested that the developer use the Symbian concept of Active Objects since it guarantees the non-preemptive multitasking mentioned in the previous chapter. An Active Object is an object that is derived from the Symbian class `CActive` and works as a single thread. When multiple threading is needed, multiple Active Objects can be instantiated. The responsibility of deciding which Active Object that is to be given time for processing, lies upon the Active Schedule*r*. The Active Scheduler is a task manager that can be responsible for as many Active Objects as needed. It is the authority that is responsible for the non-preemptive workflow, which means that any Active Object automatically is subscribing for process-time when started. This guarantee of atomicity is one of the advantages of the *Active Objects* and the main reason why using that instead of traditional threads. The introduction of the non-preemptive scheduling does however introduce the chance of objects with more important tasks will have to wait for tasks of less importance to finish. To circumvent this, each Active Object has a priority key that the Active Scheduler will consider before choosing which object to give process-time. Nevertheless it is important to make sure that the events run in Active Objects are as small as possible to avoid delays in the execution.

3.2.4 User Interface

To be able to communicate easily with a program in a dynamic manner a User Interface (UI) is needed. Symbian offers a Series60 specific high level UI framework called Avkon [17]. The two main reasons for using this is that it is faster to build and maintain than a self made low level UI framework and because it enables programmer to port programs to other Series60 models without any hassle. The Avkon framework follows the Model-View-Controller pattern [5] which urges developers to separate Views from the Model and Controller part of their programs.

3.3 Definition of the SMS

Before commencing with the program design and implementation the technicalities of the SMS protocol will here be settled. This should introduce us to the limits of the protocol itself and give us in-depth knowledge of the process of SMS transmission. This chapter will result in a theoretically expected bandwidth that we can compare against in the later Benchmark chapter.

3.3.1 The Basics

The Short Message Service (SMS) is a service that enables mobile phone users to communicate via small text based massages. The user defined part of a SMS message has a maximum length of 160 7-bit characters or 140 bytes (p.694 [8] and p.48 [7]). A standard SMS message contains no multimedia content. More elaborate version of SMS includes PSMS, EMS and MMS which offers inclusion of richer content such as audio and video. SMS messages are transferred via the control network channel, Signaling System nr.7 (SS7), and it is therefore possible to receive SMS messages whilst engaging in a voice call since SS7 do not conflict with the voice communication or Traffic Channel (TCH). This means that usage of the SS7 channel for general purposes could be implemented to be transparent to the user.

The popularity and culture that have evolved around the SMS is tightly connected to its history. Nobody had expected the popularity of the SMS and no operator therefore spend any resource on commercializing the service when it was released. The popularity of SMS sprung from the fact that the operators did not have the technology to bill SMS messages sent over pre-paid subscriptions when the service was introduced. The youth segment exploited this. By the time the operators

discovered a way to bill their users, the SMS had become an incorporated part of the users' everyday life and the service therefore remains popular today [14].

3.3.2 SMS Transfer Process

Explaining the phases a SMS message goes through in a transferring process is a very complex task. Every operator has its own internal structure that differs slightly from other operators and the set standards, which leaves generalizing such structures very difficult. In the following we will present the path of a SMS as sent over a GSM network. This network uses the SS7 control network to transfer SMS messages. The group realizes that many operators are currently upgrading their SS7 control network solutions to either hybrid SS7/IP or TCP/IP solutions but we see describing these as out of this projects' scope.

Fig.1 illustrates a simplified version of the path of a SMS message. In the following the mayor network nodes that a SMS message passes through will be presented. For detailed descriptions of every node and linkset on a SS7 network we refer to Appendix A of [14] and for an excellent OSI model description of SS7 and in depth package details see [15].

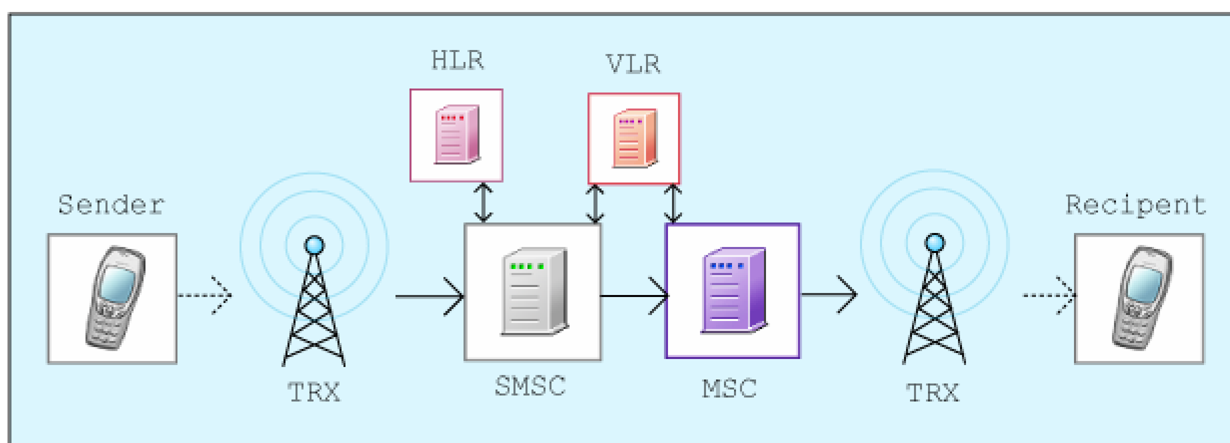

***Fig.1** The 'Store-and-forward' process of sending a SMS over a GSM network*

SMS messages are shipped in a *store-and-forward* fashion like many other text based message forms (e.g. E-mail). This means that a SMS is first transferred from the sending handset to a Short Message Service Centre (SMSC) via one or a max of 16 Transmitter/Receivers (TRXs). The SMSC is responsible for deciding how a SMS is to reach its receiver. The SMSC does this by requesting the status of the recipient through the Home Location Register (HLR). The HLR pages the recipient

phone and returns an 'active' or 'inactive' status based on whether the recipient phone is turned on or within signal reach. If the recipient is inactive, the SMSC will store the SMS message for an operator specified period of time and try to resend the message for a number of times again defined by the operator. The SMS is a *best effort* service which means that there is no guarantee that a SMS will reach its recipient if the HLR keeps returning 'inactive' statuses. If the recipient is active, the SMSC sends the message to the Mobile Switching Centre (MSC) that currently serves the recipient. The MSC then sends the message to the Mobile Server for transmission after collecting recipient information from the Visitor Location Register (VLR). The message is then sent by one or multiple TRXs to the recipient phone. The MSC finally notifies the SMSC of the outcome of the process and the SMSC then reports the delivery status to the sender.

3.3.3 Expected bandwidth

To give an initial picture of the areas that the product of this project could be utilized, an expected bandwidth of the SMS should be settled. This task is incredibly complex and relies on many variables. Variables like 'current network traffic, 'number of TRXs in current sector' and 'SMS message data per TRX time slot' all vary from operator to operator, geographical position of user and all contribute to the expected bandwidth of the SMS. In the following we will take assumptions based on data presented in [14] to finally settle what upper bandwidth is to be expected.

The channel that handles SMS transmission in a SS7 network is the Standalone Dedicated Control Channel (SDCCH). Since SDCCH is part of the control layer in a GSM network, it is assigned fewer slots per TRX time slot and therefore has a low data transfer rate. The remaining time slots are assigned to the higher priority TCH channel responsible of voice communication (Fig.3-3 of [14]). Generally 4-5 seconds are required to transmit an SMS message by SDCCH (p.17 of [14]). GSM network coverage is divided into geographically defined sectors. In [14] a typical urban sector is said to have 4 TRXs and 8 SDCCHs. Such a sector setup supports SMS traffic of:

SMS message via SDCCH = 4 sec.
SMS messages per Min. via SDCCH => 60 sec / 4 sec = 15 sms/min
SMS messages per Min. x 8 SDCCH => 8 * 15 sms/min = 120 sms/min
Max Sector SMS traffic per hour => 120 sms/min * 60min = 7200 sms/hour

With a user definable body of only 140 bytes per SMS this gives us a bandwidth of a mere 35 bps.

It is hard to make any generalization about bandwidth requirements to handsets if streaming of audio and video should be possible. This is based on the fact that every handset has their own CPU, RAM and screen resolution specifications. According to the above settled bandwidth a lot of usage possibilities can be ruled out from the start despite the differences of handsets. Streaming of both audio and video would be impossible via SMS as would any other high priority data transfer. Transferring 1Mb of data would take about 8 hours!

The group still sees possibilities of utilizing the exploit of SMS despite the discouraging information above. One can imagine synchronization of contacts and calendars between project participants' phones and low priority data transfer of e.g. ring tones. Remember that users might accept the long time of waiting if a service is free.

Further more, we can reasonably hope for higher bandwidth benchmarks since we expect that most of today's operators use hybrid SS7/IP solutions as mentioned in the intro of the chapter. Since the SS7 control network date back to the early 1990es most operators should have somehow updated by now. This means higher SMS traffic support and therefore higher bandwidth. The later Benchmark chapter will prove us right or wrong in this matter.

3.4 Ensuring Quality of Service

3.4.1 Reliability of the SMS service

If a file should be exchanges between two Smartphones using the SMS, some segmenting of the file would be necessary. Remember that each SMS can only hold 140 bytes. For a 1Mb file this would result in 7143 chunks of 140 bytes being transferred by SMS. This immediately poses the question regarding reliability that was posed in the problem statement. While researching the SS7 protocol the group found statistics in [1] that ensure us that measures should not be taken to improve reliability. The article promises, with a 99,99.. percent probability, that SMS' will be delivered in sequence and loss free. We will therefore not take any initial steps to ensure 'in sequence chunk assurance' or 'lost chunk retransmission' in the design chapter.

We have later discovered research in [14] that examines maximum load on all elements of SS7 in cases of terrorism. These introduce circumstances under which SMS messages will be lost. Such low probability circumstances should be handled in a release version of our program but will be disregarded in this prototype build.

3.4.2 Validating files

In regards to streaming data, the loss of a chunk or two while listening to music or watching video would not matter much. But in most other cases it is of utmost importance that the entire file is received and reconstructed at the receiving end to avoid corrupt files. This issue regarding validating files was stated in the problem definition. We again reefer to [1] regarding the 99,99.. probability of messages arriving in sequence and loss free and will therefore not deal further with the issue of validation. If we reconstruct a file in the order the SMS stream is received no problems should occur.

The special terrorism cases from chapter 3.4.1 apply to the issue of validation as well and will be disregarded in this project.

3.5 Encoding before Transmission

3.5.1 Compressing data

In most cases of data streaming, some level of data compression is a very good idea before transmission. In video and audio streaming such compression (sending peer) and decompression (receiving peer) is often done real-time. In the prototype implementation of this project, a similar compression scheme would be a good approach to reducing the number of SMS messages a file should be broken into. At an early stage we therefore started looking for open source Symbian compression libraries. In this search we discovered the `CZipFile` class, based on the original PKZip algorithm, of the Symbian SDK which has been supported since Symbian v.7.0. If time allows it, we will research the gain of using this class and implement a compression scheme.

3.5.2 Encrypting data

Protection of privacy and sensitive data is a matter to be considered in most networking applications that are designed to send data files. This gives rise to a need for data encryption (p.39 of [14]). In our initial research we therefore went ahead and looked for open source libraries that could aid us. We soon found that that the Symbian SDK also supports encryption. More specifically, the `RencryptStream` class supports symmetric cipher based encryption [18] of data streams. If time allows it, the group will utilize this encryption class and implement our application to support optional encryption.

3.5.3 Preserving Battery Time

Any form of data encoding usually takes a considerable load of calculations to perform. When developing applications for mobile devices the gain of such operations should be carefully considered, since battery time is limited. Would compressing data before sending e.g. be more battery consuming than sending the data as it is? In a release production of an application like this one, optimal preservation of battery time should be tested and implemented. The group will not spend time on such tests since this is a prototype production.

4 Program design

Before starting program implementation we needed to agree on a program design that includes all functionality necessary to carry out the operations stated in the Idea chapter. This chapter introduces and describes every aspect of the program to be implemented as well as the programs life cycle. This chapter will result in a UML class diagram that we will use as reference throughout the implementation phase.

4.1 Choosing a file

In order to send a file we need to choose one from the operating file system. This can be done by either implementing a file browsing class or by hard coding paths to predefined files. For testing purposes the group finds it satisfying to use four different predefined files containing different media types and being of different sizes. The difference in media types gives us an increased opportunity to discover any divergence in the files before and after transmission. A single package lost in a sequence of more than 20.000, which an MP3 file easily can be, may not be discovered when listening to the music after transmission. A single package in a document containing text will with greater probability contain information crucial for the receiver. The difference in file sizes is important to give a picture of how the program reacts in different situations, e.g. a memory leak may not appear before processing larger files.

4.2 Dis-assembling a file

Since an SMS message only contains 140 bytes and most files are many times that size, the files need to be disassembled into minor pieces, that we call *chunks*, before transmission. A class of our program should take care of this disassembly. In order to be able to recognize chunks of a file when they arrive at the receiver, these should all be marked with a data pattern in the beginning of the message that we call the *ProgramTag*. The first chunk produced will be used as a header chunk and contains information that tells the receiver that a new chunk transmission sequence is starting, the name of the file and how many chunks to expect before finishing file assembly. The following chunks will contain file data.

The class `CChunkGenerator` will take care of the above.

4.3 Sending a message

In order not to overload program heap or stack, chunks of a file should be sent instantly and not when a whole file has been disassembled. The `CChunkGenerator` class will therefore pass its generated chunks directly to a class responsible of sending the chunks by the SMS. This 'sending' class will construct a SMS and pass it to the Outbox for transmission, as fast as possible. Since we will not re-send messages to assure arrival, we can delete the message after handing it to the Outbox. The sending class should then clean up any trace of the sent SMS message. Doing so will minimize the chance of the application running out of memory.

The class `CSmsByMtm` will take care of the above.

4.4 Waiting for and receiving a message

The Symbian OS will not automatically detect that an incoming message is of interest to our program. Our program will therefore have to listen for messages at any given time. This will be done by listening for new entries in the Inbox folder and check incoming messages against a number of message rules before deciding whether to process a message further. The message rules that will be enforced are first of all that messages are of the type SMS message. Any messages that are e.g. of the type MMS or e-mail can be classified as not of interest and be handed over to the operating system. Secondly the messages have to contain the *ProgramTag* of our program. If these rules are fulfilled the content of the message will be extracted, passed to another class for file assembly and the message deleted.

The class `CSmsSessionObserver` will take care of the above.

4.5 Re-Assembling a file

Since we presume that the messages arrive in sequence, the messages arrive in the same order that they are intended to be assembled. Once a message receiving sequence has been initiated a class should be responsible for reassembling received chunks into a file. The first chunk message, the header chunk, contains file name and total numbers of chunks. This message starts the assembling process by creating a new file in the file system that will continually written to until the total number of chunks have been processed..

The class `CAssembleToFile` will take care of the above.

As mentioned in the Technical Analysis we wanted to separate the tasks performed by the individual services to avoid conflicts and maintain atomicity. The final event diagram after designing how the program should process the files can be seen in fig. 2.

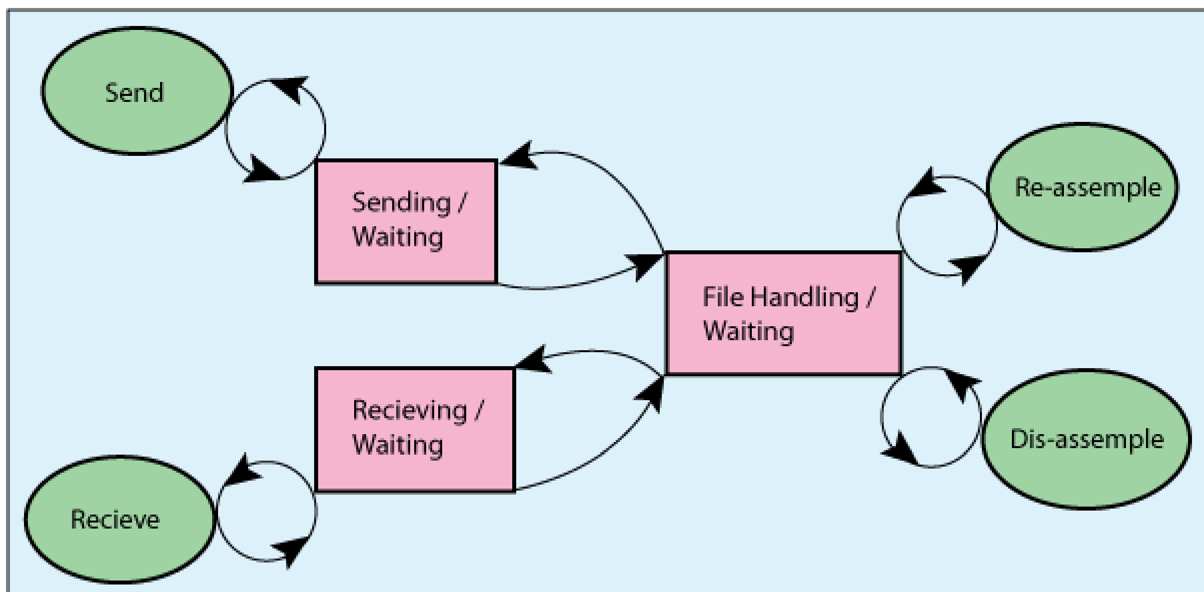

Fig.2. *Event diagram. Services shown in squares. Events shown in circles.*

4.6 Timing the process

To get benchmark results from the application we are going to time different processes related to sending a file by SMS. We are going to benchmark the time it takes to disassemble and send a file from the sending phone and the time it takes to receive and reassemble a file at the receiving end. In

addition we will test how long time it takes the network to transfer messages of 140 bytes between phones. This will give us benchmarks for both bandwidth and processing for both sending and receiving phone.

The class `CTimeStamp` will take care of the above.

4.7 UML class Diagram

From the description of the program design and the needed functionality we created a class diagram (see fig.3) to support the implementation process. An object oriented design is not the objective of this project but a way to create easy maintainable and well-arranged code. Principles of OOD have therefore been followed in our program design.

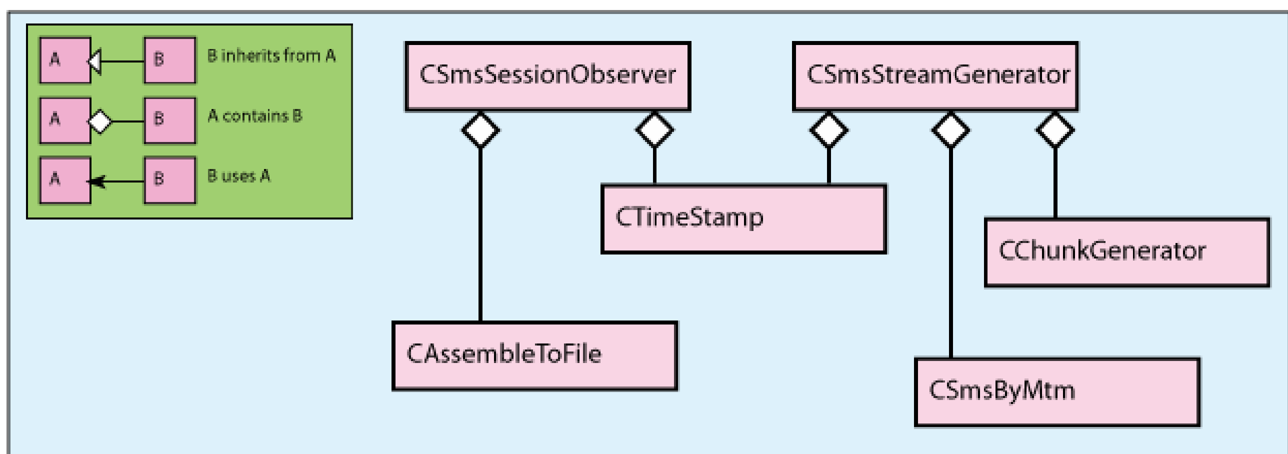

Fig.3 *Initial class diagram. Notice the simplicity of the prototype.*

5 Implementation

5.1 Symbian; Getting acquainted

Being comply new to Symbian OS development, the group found the first week of implementation a very steep hill to climb. Our lack of experience in C++ do not make learning a new variant of the language come to us naturally. This chapter will summarize some of the points that we find to be unacceptable or just incredibly annoying when starting to use Symbian C++.

5.1.1 Naming Conventions

Symbian C++ introduces it own naming convention (p.19 in [9]). A very good initiative the group must agree but also a bit difficult to get used to. It is easy to forget these conventions when using them the first time. Code does get easier to scan when using these conventions, the group later realized.

Another weird thing the group noticed was the number and form of all the enumerators and constants of Symbian C++. First of all there is a huge amount of these. Secondly they are very long which must have been found necessary since there are so many. Getting used to these 30+ character constants and telling them apart took time getting used to.

5.1.2 Where is the console?

Programmers need to test their applications some how. Simplest way to do this is to generate output at certain points of interest in your code. When this group after a few ours of development came to a point where we needed such a feature, the question that entitles the section popped up. Symbian does not to our knowledge offer a console to print output to. Standard C++ and Java offers this feature that we have come to take for granted, so why doesn't Symbian? The fastest way to get the desired output was to print this on the screen of the SDK emulator through the Code Warrior IDE. We achieved this by implementing the `CScreenTextView` class in our program. This class has been a key element throughout our development.

We do realize that classes and methods exist that enables us to do pop-up windows or alert boxes with text we can define ourselves. These methods either require user input to cancel or take a certain period of time to go away. These were therefore found useless in this case.

5.1.3 The time waist of emulation

Having some experience with J2ME programming, the group is used to testing programs on phone emulators in a matter of seconds. This is not the case when using Symbian SDKs. Starting up an emulator takes about 1,5 minutes. This results in 60+ minutes of waiting each day of the implementation phase. Very annoying.

The group tried installing older versions of the Symbian SDK hoping that old versions, lacking features that we were not using, would start up faster. This was not the case.

5.2 Class descriptions

In the following we will go through the classes of the project. The classes are divided into the sections they belong to, which are: *foundation*, *timing*, *views*, *file handling* and *messaging*. The description of each class will be limited to methods and code snippets that we find especially important for the project. If other parts of the code are of interest to the reader we reefer to the documented full source code that can be found in Appendix I. For the sake of overview a final class diagram is provided in fig.4 below.

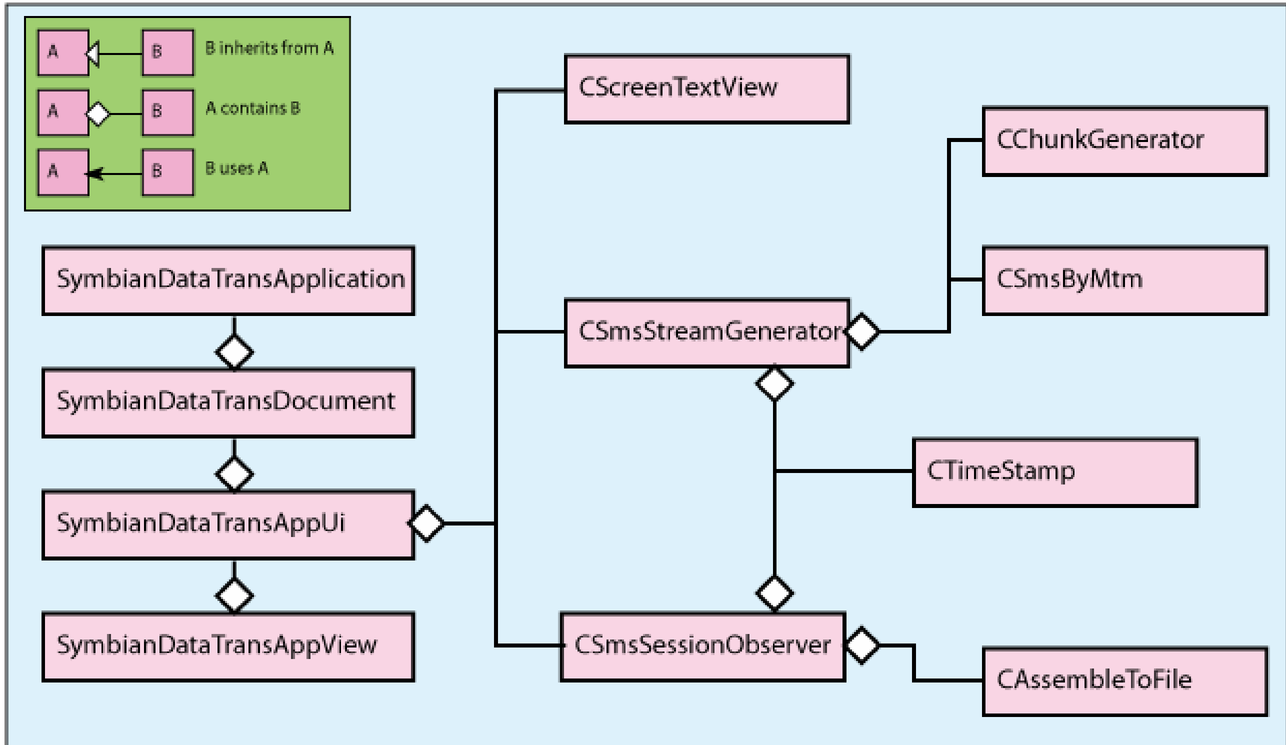

Fig.4 *The final class diagram.*

A general comment regarding our implementation; all implemented classes support two-phase-construction even though such are not necessary in some cases since object construction involves no leaving calls. We did this in case the need for two-phase-construction would pop up during development.

5.2.1 The foundation

The basis of a Symbian C++ GUI implementation consists of four basic classes (p.99 of [9]), each having their own responsibility in launching the program. Because of the memory limitation of the Series60 and the need to reduce the start up time of applications, these classes only perform the most necessary tasks. For other platforms such as the UIQ [19] the foundation classes is used for more demanding tasks such as file creation and opening. Due to their limited use in this project they will only be described shortly in the following.

`SymbianDataTrans`

The DLL entry point of the implementation. Constructs the `SymbianDataTransApplication`.

`SymbianDataTransApplication`

The application class specifies properties of the program. In our program the only property is the UID (unique program ID). The class constructs the `SymbianDataTransDocument`.

`SymbianDataTransDocument`

The document class is responsible for creating the application UI. Constructs the `SymbianDataTransAppUi`.

`SymbianDataTransAppUi`

The `SymbianDataTransUi` class is responsible of creating the views of the application and handling of commands such as key presses. Many of the Symbian applications that we have encountered during this project implement a lot of their core in the AppUi class. We find this to be bad practice and have therefore deliberately separated as much as possible for the AppUi class to achieve a more modular design.

In our program the AppUi class handles our simple menu system and the key presses related to this. As mentioned earlier we use hard coded file paths and recipient addresses instead of a file browsing and dialog system. The hard coded file paths and recipient address is contained in `SymbianDataTransUi`.

5.2.2 Timing

`CTimeStamp`

The `CTimeStamp` class is used for benchmarking purposes. It provides a `Stamp()` method for marking the current time using a `TTime` object to fetch the system time. The `GetTime()` method sets a second time mark and returns the time interval in milliseconds between two time stamps. Furthermore the class provides a `GetBandWidth()` method for calculating a bandwidth based on a time intervals and a file size.
The `CTimeStamp` class is used in classes `CChunkGenerator` and `CAssembleToFile` to calculate the sending bandwidth and receiving bandwidth.

```
TInt32 CTimeStamp::GetBandwidth(TInt fileSize)
{
  //has iStart and iEnd been set?
  if(iTimeSpan>0)
  {
    return (fileSize*1000)/iTimeSpan;
  }
  return -1;
}
```

Code Snippet 1. The `GetBandwidth()` *method of class* `CTimeStamp`*. FileSize is scaled by factor 1000 before dividing with time span integer to avoid clamping.*

5.2.3 Views

CScreenTextView

Since the program is not very dependent on complex UI we only make use of a single View class. This class is responsible for updating a heap residing `HBufC` descriptor either by replacing its data via the `SetScreenText()` methods or by appending data to it via the `AppendScreenText()` methods. The class inherits from `CCoeControl` and uses its `Draw()`method to access the screen canvas and draw the content of the `HBufC` descriptor to screen.

Our `CScreenTextView` class lacks some functionality but solved the needs of this project fine. The functionally lacking is auto formatting of text and support of the newline character '/n'. We later learned that a `CRichText` object must be used to gain such behavior. A newline function was hard coded to enable multiple lines of text.

```
TInt lineIndex = 1; // the line number we are currently using
TInt txtSize = iScreenText->Length(); // nr of txt chars
TInt blockSize = 25; // nr of chars we allow on each line
TInt charEnd = txtSize; // how many chars left on current line
TPtrC16 tmpText = iScreenText->Right(txtSize);

for(TInt i=0; i<txtSize; i++)
{
  if(i==(blockSize*lineIndex))
  {
      TInt baseline = (font->AscentInPixels()*lineIndex)
```

```
          +(6*lineIndex);
      gc.DrawText(tmpText.Left(blockSize), rect, baseline,
          CGraphicsContext::ELeft);
      charEnd-=blockSize;
      lineIndex++;
      tmpText.Set( tmpText.Right(charEnd) );
    }

}
```

Code Snippet 2. Part of the `Draw()` *method of class* `CScreenTextView`. *Newline behavior is achieved by looping and iterating text.*

5.2.4 File handling

`CSmsStreamGenerator`

An application as the one dealt with in this project could be designed to make future message type support easy to develop and incorporate. If the classes responsible of sending messages were designed in a modular manner, it would be easy to choose which sending module to use if some module managing class were present. We have therefore implemented the `CSmsStreamGenerator` class that manages how to send messages by selecting a sending module. The class will start a SMS stream process and send a stream of SMS messages using the currently selected sending module. Using the principle of the `CSmsStreamGenerator` class, it would be fairly simple to incorporate other message type support (e.g. MMS messaging) into our application.

The idea and reason for implementing a module managing class originally sprung for the fact that we intended to use multiple sending techniques to transfer SMS messages. We would therefore have two different sending modules to manage and a managing class was therefore needed. We later cast the idea of a multiple sending solutions aside as will be explained in the 'Experiments and Revelations' chapter.

The `CSmsStreamGenerator` class starts a SMS stream process by calling its `GenerateStreamL()` method that passes a file path to the `CChunkGenerator` instance it owns. This `CChunkGenerator` instance disassembles a file into chunks which are then sent via

CSmsStreamGenerator:: SendSmsL(). The SendSmsL() method sends messages using the currently selected module as seen below.

```
//send SMS by set module
TBool CSmsStreamGenerator::SendSmsL(const TDes& SMS, const TDes&
recipentNumber)
{
  if(iTrans==ETransferByMTM)
    return iSmsByMtm->SendSmsByMtmL(SMS,recipentNumber,
                                    KSMSCAdress);

  //if(iTrans==ETransferBySocket)
  //return iSmsBySocket->SendSmsByMtmL(SMS,recipentNumber,
  //                                  KSMSCAdress);

  return EFalse;
}
```

Code Snippet 3. The SendSmsL() *method of* CSmsStreamGenerator. *Messages are sent using currently selected module.*

CChunkGenerator

The CChunkGenerator class is responsible for splitting a file into chunks of 140 bytes and sending these via the CSmsStreamGenerator reference it owns. A SMS stream transmission session is started by calling WriteFileToSmsStreamL(). The method first creates a header chunk that contains KProgramTag to identify our program, total number of chunks the file is spilt into and finally the path of the file. This initial chunk is then sent and will instantiate a SMS stream receiving session on the recipient handset. From this point the class reads 140 byte of the chosen file at the time into an HBufC8 descriptor, which is then passed to the sending method. This continues until the entire file has been transmitted.

To ensure the file splitting process worked, the group implemented the ReadFileFromSmsStreamL() testing method. If the WriteFileToSmsStreamL() method is told to pass its created chunks to the ReadFileFromSmsStreamL() method instead of the message sending methods, it will reassemble the disassembled file. We used the filesAreEqualL() method to verify the reassembled file. The reassembly test was a success and proved our method's functionality.

The WriteFileToSmsStreamL() method writes session progress and timings to screen using a CTimeStamp instance.

CAssembleToFile

The CAssembleToFile class is responsible of reassembling a SMS stream at the receiving phone. It does so using its DoAssemblingL() method who's design is inspired by the CChunkGenerator::ReadFileFromSmsStreamL() testing method.

The class writes receiving end timings and bandwidth output to screen using a CTimeStamp instance.

5.2.5 Messaging

CSmsByMtm

This class is the main sending module of the project. It sends SMS messages using the Symbian MTM framework (p.321 of [9]). To utilize the MTM framework the class must inherit from MMsvSessionObserver to be able to create a *message server session* and therefore does so. A message server session is opened as part of a two-phase construction as shown below.

```
void CSmsByMtm::ConstructL(CSymbianDataTransAppUi* aAppUi)
{
  iMsvSession = CMsvSession::OpenAsyncL( *this );

  iAppUi = aAppUi;
}
```

Code Snippet 4. A CMsvSession *is opened asynchronously in the* CSmsByMtm::ConstructL() *method.*

Calling methods from a message server before the opening process has completed will of course crash a program. This means that in order to utilize any methods of our CMsvSession* instance we must wait for the opening procedure to finish. Luckily the OpenAsyncL() method returns an EMsvServerReady event once the message server is ready. We catch this event in the inherited

HandleSessionEventL() method and complete construction as shown below. Kind of a three-phase-construction one could say.

```
void CSmsByMtm::HandleSessionEventL( TMsvSessionEvent aEvent,
                                    TAny* aArg1, TAny* aArg2,
                                    TAny* /*aArg3*/)
        {
        switch ( aEvent )
                {
                case EMsvServerReady: // Session established
                {
                        CompleteConstructL();
                        break;
                }
```

Code Snippet 5. Completing construction after server session have been established in CSmsByMtm::HandleSessionEventL()

Once the three-phase-construction is done all objects needed to send SMS messages have been instantiated. The method SendSmsByMtmL() is responsible for doing so. To explain every aspect of sending a SMS message using the SMS MTM would result in a very extensive chapter. We will therefore here describe the main parts of the process and reflect on choices taken during development.

To send a SMS message a new *entry* must first be created in the 'Drafts' message folder. The SMS message will reside in this folder whilst under construction. A message is created in drafts using the MTM framework class CSmsClientMtm. After the creation of the SMS message, the content and settings is set. The content passed from CChunkGenerator is put into the message body and the recipient and SMSC address is set using the MTM framework class CSmsSettings. After this, message construction is complete and the message is moved to the 'Outbox' folder and then scheduled for transmission.

Moving and scheduling messages are CMsvOperation operations. Such operations are handled asynchronously unless told otherwise. In our program, problems occur if we let the Active Scheduler of the Symbian OS handle the order in which these operation are carried out. First of all, our program runs out of memory if we do not ensure that the operations are carried out one at the

time. Secondly, the order of these operations is important if things should work as expected. The group therefore had to enforce that the operations were handled synchronously. We did so using a `CMsvOperationActiveSchedulerWait` object. This object is a 'mini thread' that can be used to halt the program until a `CMsvOperation` is complete as shown below.

```
CMsvEntry* parentEntry = CMsvEntry::NewL(
         iClientSmsMtm->Session(), aEntry.Parent(), selec );
CleanupDeletePushL(parentEntry);

CMsvOperationActiveSchedulerWait* waiter =
         CMsvOperationActiveSchedulerWait::NewLC();

CMsvOperation* aMoveOpp = parentEntry->MoveL( aEntry.Id(),
         KMsvGlobalOutBoxIndexEntryId, waiter->iStatus );

waiter->Start();

CleanupStack::PopAndDestroy(waiter); //  waiter
CleanupStack::PopAndDestroy(parentEntry);    // parentEntry
```

Code Snippet 6. Enforcing synchronous handling of `CMsvOperation` *using* `CMsvOperationActiveSchedulerWait`.

Once a message has been sent, it is automatically moved to the 'Sent items' folder by the Symbian OS. Since our program should leave no trace of how its operations are carried out, it is our responsibility to delete message belonging to our program. We do this in two ways. First, the sent items folder is cleaned of messages belonging to this program every time the program starts up via the `DeleteAllSentByProgramL()` method. This ensures that no messages from prior sessions linger in the folder. Secondly we delete messages by catching message handles in the `HandleSessionEventL()` method every time a message is moved. The event handler method receives an `EMsvEntriesMoved` event when this occurs. By checking if the moved messages reside in the sent items folder and that they carry our program tag, we can delete them and do so.

CSmsSessionObserver

The `CSmsSessionObserver` class listens for incoming messages on the receiving handset. Like `CSmsByMtm` it inherits from the Symbian MTM `MMsvSessionObserver` class which provides an interface for notification of message related events. The events that are important to this class are

EMsvEntriesCreated and EMsvEntriesChanged. These are caught by the HandleSessionEventL() method as in CSmsByMtm.

Once an incoming message begins downloading to a handset an EMsvEntriesCreated event occurs. We can catch this event, check if the new entry is in the 'Inbox' and get access to the entry. Only problem is that even though the new entry is available for access, it might not have any content yet since it might still be in the process of downloading. When an EMsvEntriesCreated occurs we therefore only fetch and store is ID.

```
case EMsvEntriesCreated:
{
  TMsvId* entryId = STATIC_CAST(TMsvId*, aArg2);

  //Is new entry in inbox?
  if(*entryId == KMsvGlobalInBoxIndexEntryId)
  {
    CMsvEntrySelection* entries =
         static_cast<CMsvEntrySelection *>(aArg1);

    //get new entry ID
    iNewIncommingID = entries->At(0);
  }

  break;
}
```

Code Snippet 7. Storing created entry ID for later use in the EMsvEntriesChanged *event.*

When a EMsvEntriesChanged event occur we again check the entries residing in the inbox and then find the entry matching the ID stored when the EMsvEntriesCreated event occurred. By doing so we are now sure that the entire message is available and can begin processing it.

Final task of the CSmsSessionObserver class now that we have a new message belongs to our program in selection, is to extract the SMS message body, pass the body to CAssembleToFile for reassembly and finally delete the message from the inbox. Extracting the message body proved to be a bit of a hassle. A SMS message body is of type CRichText and getting the content of a such into descriptor form was not strait forward. Research told us that the method to be used to

extract content of a CRichText vary from one handset to another. Very strange. We managed to implement an extraction method that works on all testing handsets of this project. This method is shown below.

```
//fetch message body
CRichText& body = smsMtm->Body();

//put body into descriptor
HBufC* bodyBuffer = HBufC::NewL(body.DocumentLength()) ;
CleanupDeletePushL(bodyBuffer);
TPtr bodyDes = bodyBuffer->Des();

body.Extract(bodyDes, 0) ;

//is message tagged by our program?
if( bodyDes.Left(4).CompareC(KProgramTag)==0 )
  // send sms to assembling
  iToFile->DoAssemblingL(*iAppUi->Convert16To8BitL(*bodyBuffer));
                                    }
```

Code Snippet 8. Extraction content of a CRichText *to a descriptor.*

After passing the message content to reassembly the message entry is deleted in the same manner as the deletion methods of CSmsByMtm.

To test if the reassembly process above works, we implemented a little hack that supports offline reassembly. This was inspired by the similar testing done in CChunkGenerator. By *moving* new messages created in CSmsByMtm to the *inbox* instead of moving them to *outbox* and scheduling them for sending, we can catch these 'new' messages in CSmsSessionObserver by watching the EMsvEntriesMoved event of the event listener. By doing so we tested that the reassembly of files from a SMS message stream generated by CSmsByMtm works.

By doing so we proved our concept to be possible!

5.3 Experiments and Discoveries

Building the application of this project involved a lot of experimenting with Symbian C++ functionality as well as third party software. Even though we had done some preliminary work such as building a class diagram we still had a lot of uncertainties regarding implementation specific areas such as network communication protocols, testing environments, the programming language itself and the operating system architecture. This chapter goes through some of the experiments and the revelations.

5.3.1 LeaveScan

Scanning [11] we got acquainted with the tool *LeaveScan*. This program scans a Symbian application for methods that might and might not 'leave'. Following [22] we integrated LeaveScan into the Code Warrior IDE and used the tool throughout the project to identity and properly name leaving methods.

5.3.2 Socket communication

The Symbian Operating System provides a wide range of communication protocols to support an even wider range of communication infrastructures, including Bluetooth, TCP/IP, IrDA, USB, MMS and SMS. One of the technologies we experimented with when communicating between devices was sockets. A socket consists of a physical machine network address and a logical port number. This combination is unique on any given network and thereby allows an application to identify another location on the network with which to communicate.

We made a test application that was running on the phone and served as both client and server. The client part of the application used the telephone number of the receiver as the network address with a specified port number. The server at the recipient then could listen for communication on that specific port number.

Another way to do the socket experiments is to use the same port number as SMS messages normally use in the Symbian OS and then pick the messages from the inbox. After working with such a socket approach for some days we realized that it was not the simplest way to communicate

between two handsets. The communication channel could be established much easier if we chose to use the built-in messaging components (MTM base classes) that Symbian OS provides. To verify that we were correct in our assumptions regarding the simplicity of using the MTM, we send emails to the heads of Symbian courses at all the technical universities of Denmark as well as emails to aboard references supplied by our supervisor. The answers we got all advised us to use the MTM and we therefore did so. The source code for the socket experiment can be found in the Code Warrior Project under 'Code' on the attached CD. Look for the `RSmsBySocket` class in the 'Trash' folder under 'src' and 'inc'.

5.3.3 The Nokia Connectivity Framework

The Nokia Connectivity Framework [12] enables developers to emulate communication over many network protocols between multiple emulators. This should help us send SMS messages from one emulator to another for free and hence function as our main testing environment. Supposedly that is. The group spent a very long time trying to get this tool to work with Symbian emulators. We can get the tool to work with J2ME programs but not with Symbian ones, even though the NCF documentation states that it supports this. Since the NCF is the only tool of its kind, we have been able to find, it has been the only way to emulate the process of sending and receiving SMS messages software-wise. We have contacted all Forums we could find, have asked all Symbian references supplied by our supervisor and have sent emails to teachers of Symbian courses on the technical universities of Denmark without any luck. What this basically means is that we are without a software testing environment and that the only way to test our program that we are left with, is to run our program on the real testing phones of the project. Since none in the group has a 'free SMS' subscription this fact limits the size of the files we will transfer for testing purposes due to the cost of this.

5.3.4 Porting the Program

Having embraced the fact that SMS software emulation would be out the question, we then turned our attention towards creating a program build that could be installed and run on the actual testing phones. This involves creating a SIS file as in [13] and transferring this file to the phone for installation. During this process we stumbled upon strange behavior. After installing the SIS file on the phone, the application was nowhere to be found. Code Warrior would report only warnings but no errors while compiling so we found this very weird. We quickly concluded that this fact could be

related to the libraries we where linking against in out MMP file. We therefore searched our entire program to make sure that every aspect of it was supported by our testing phones. This did not change anything. After this we turned to more experienced developers on the forums of www.newlc.com who, from the only warning feedback given from our compiler:

```
tool exit status == -2
Dll 'SYMBIANDATATRANS[03D20F8A].APP' has initialised data.
```

Code Snippet 9. Warning received from compiler due to static data members in our program.

could tell us that our problem was due to the usage of global variables or static content inside methods. Since Symbian programs are actually DLL's they can not have writable static data. We did in fact have static data in a method at this point and redesigning this method made our program work on the phone.

A few reflections should be made upon this experience. Why does the program work on the emulator but not on the phone? Even though the group has experience with the fact that programs running on an emulator might act differently on phones from J2ME development, we find it odd that the Code Warrior compiler does not report errors when static members are wrongfully used in a program. Especially if it is such an essential part of Symbian development. We furthermore find it very strange that we have not come across anything relating to this in our text books.

A little side effect of the above research was an accidental discovery of the content of the *.APP.MAP file that resides in the same director as a built *.APP file after doing a THUMB UREL build. In the bottom of this file one can see which of the *.lib files included in a project *.MMP file is and isn't used. Very Nice.

5.3.5 Low Memory Tool

To identity memory leaks, the group spent time researching the *Low Memory* tool found at https://www.symbiansigned.com. This tool can automatically runs stress tests on installed applications and returns the number of memory leaks that can be identified in a program. The program of this project revealed a single memory leak that we despite extensive usage of the cleanup stack have not been able to identify.

5.3.6 Full Outbox Crash

When we started testing the process of sending SMS messages we began getting program crashes (see fig.5) that we found very hard to identify the source of. By coincidence a member of the group had been roaming the 'Epoc32' folders of our emulator that day and noticed that the 'outbox' folder of the emulator contained exactly 256 entries. With this in mind we inspected the strange program crash again and found out that the program would crash when exactly 256 messages had been sent. We therefore saved a lot of time trying to debug this ghost error by a matter of coincidence.

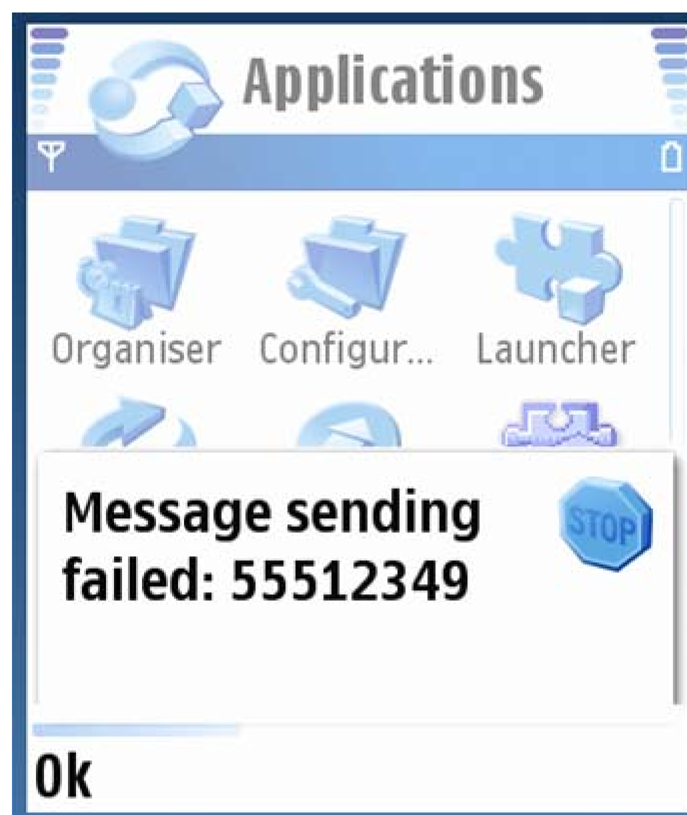

Fig.5 *Error received when 256 messages is sent using the emulator.*

5.4 The Final Prototype

We have developed a program according the lines drawn in the program design chapter. We have added extra classes to support on screen feedback and future messaging modules. We have experimented with third party software to detect syntax terminology errors and memory leaks. We have furthermore proved the concept of SMS file transfer possible by simulating a sending process locally on the phone via emulators. Everything is now lined up for over the air (OTA) testing.

5.4.1 The first OTA test

Three days before turn in we ported our program to two of our test phones by Bluetooth, transferred our test files in the same manner, opened our program on both phones, started a SMS stream session and took a deep breath. The messages were successfully sent from the sending phone. The receiving phone was less successful. After receiving the initial header message and successfully starting a receiving session, the receiving phone crashes before having received all files chunks with a probability of 50%. This is not a satisfying result. By repeating the above procedure and analyzing the phones inbox, we leaned that SMS messages do not arrive in sequence as [1] promise us with a 99.99.. probability. This news was very unwelcome at that late a stage of the project. The news was especially disappointing because we had planned various chunk verification and resending schemes at early group meetings before we found [1]. The group at that point decided not to spend the remaining time on hacking the problem introduced by packages arriving out of sequence. This choice was based on the fact that we had proved our concept via local emulator testing and had a working prototype. Only unforeseen handling of SMS messages by our operator have resulted in poor test results. We realize that many solutions to 'out of sequence' problem exists but will leave the discussion of these to be the subject of the oral project exam.

6 Benchmarks

The following chapter contains the test results we gained from testing the application. The application is tested on both real phones and by running the emulator. The most valuable results we got comes from testing the application on real phones. Unfortunately such testing requires a lot of SMS messages being sent. For financial reasons, the numbers of tests performed, is therefore limited to sending a 486 byte file 23 times. Testing the application on the emulator gives a result that varies depending on the size of the PC running it. Therefore it is not found reliable as a tool for measuring real world results but as a tool for testing correctness of code. Furthermore we did not have access to a test environment that could emulate network communication between two phones and the emulator test is therefore only a test of local processes.

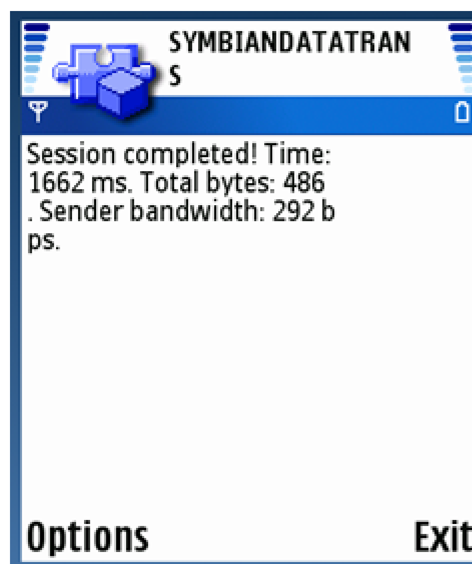

Fig.6 *Screen report in emulator after sending a 486 byte file.*

The test data that the figures of this chapter is based on can be found in Appendix II.

6.1 Testing on mobile phones

For testing the application on mobile phones we used a Nokia 6600 [20] as the sender and a Nokia 7210 [21] as the receiver. The order of sender and receiver was not reversed due to our limited access to free SMS messages, even though the two models are almost identical regarding hard- and software, the order might be responsible for a part of the differences experienced when timing the sending/receiving processes. The two phones are subscribers at two different mobile phone

operators; this may inflict longer processing time than sharing an operator because the messages need to be passed between networks.

The test is performed using a file on 486 bytes which gives us a total sequence of one SMS header containing metadata and 4 SMS messages containing the file. During the test we found that there was no guarantee that the messages arrive in the sequence they are sent. Since the program expects 140 bytes in each message but the last, it crashes if the last message arrives in the middle of the sequence. This is the reason why we have less test results from the receiver than the sender.

6.1.1 Processing messages at sender

The points of measurements seen in Fig. 7 from processing data at the sender's phone reflects the time it takes to create the header message and split the chosen file into smaller parts before sending it. As the readings shows, 21 of the 23 times the file was sent, the processing time ended within a range of 6000 to 7200 ms. This shows what we already expected, namely that the operating system has a very continuous load and the time it takes to process a file can be estimated with some certainty.

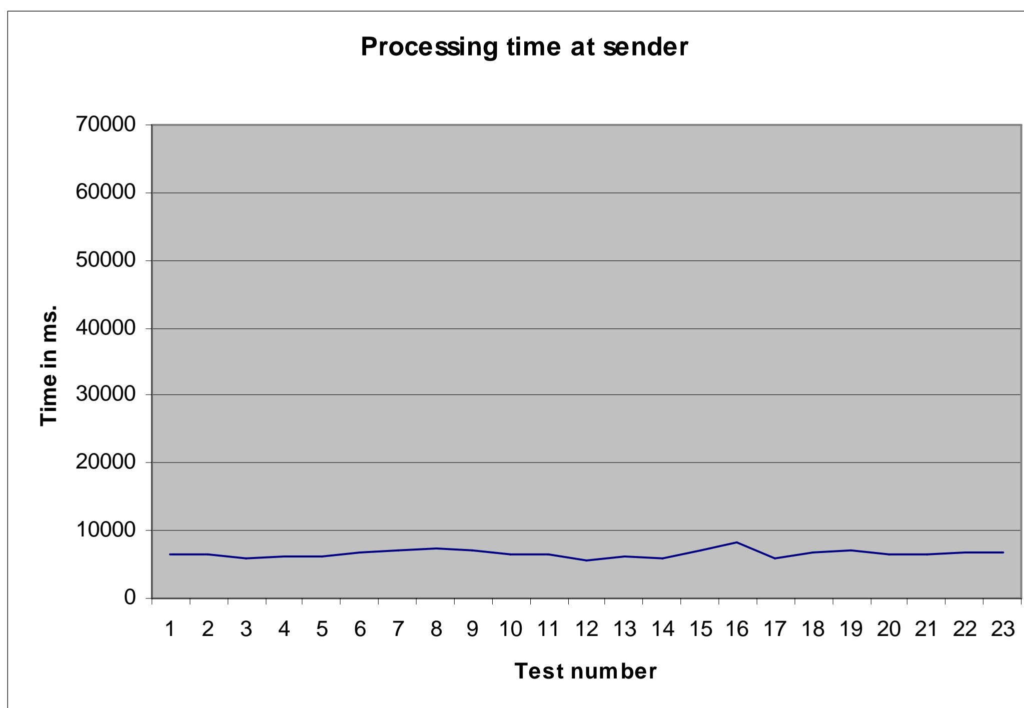

Fig.7. *The time it takes to process a 486 byte file on the sender's phone. The same process measured 23 times.*

When we calculate the local processing bandwidth of the sender we that we can expect the program to process between 70 and 80 bytes per second on a Nokia Series60 model. This may not seem as a lot considering a situation were several megabytes of data needs to processed, on the other hand it is in the current situation more than enough to create the data packages faster than the phone is able to send them.

6.1.2 Processing messages at the receiver

What we can see from the measurements in fig.8 is that processing the incoming data on the receivers phone takes almost 10 times as long for the program to write data to the file as it does reading it. As mentioned earlier one of the reasons can be differences on the hard- or software platform on the two phones. Another contributory reason could be that it is simply more demanding for the operating system to write than read due to file system rules about read and write access. When looking at the Fig 7 and 8 we can also see that there is a distinct distance between the peak and bottom positions on the graphs. This could mean that the operating system is busy carrying out other tasks such as listening for and receiving the next message. This could be a third reason for the increase in processing time.

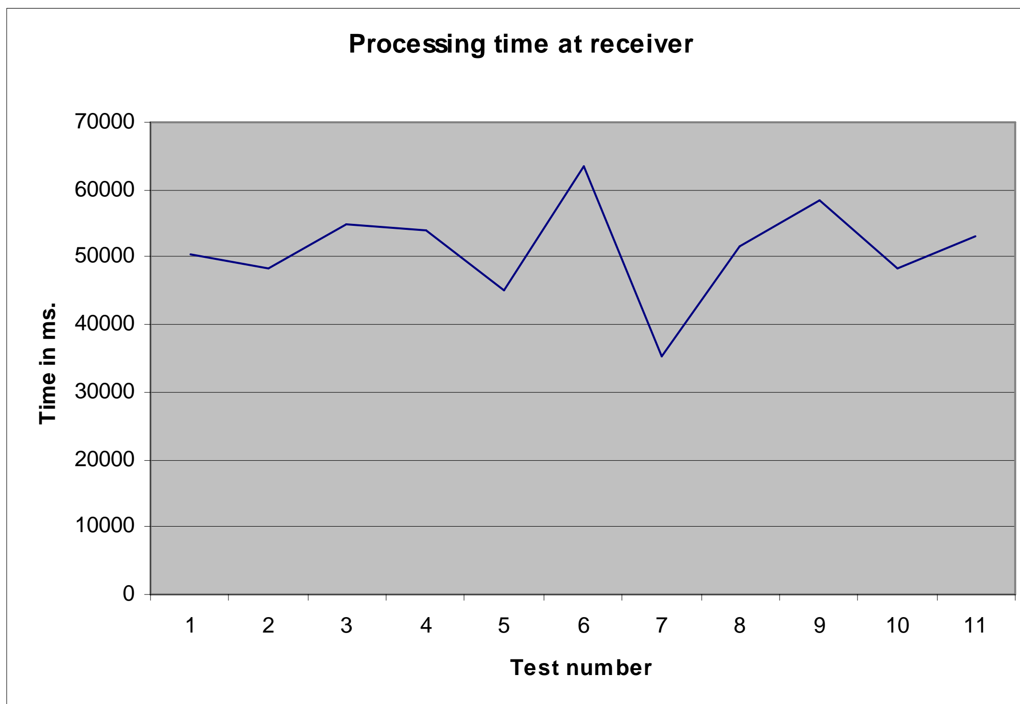

Fig.8. *The time it takes to process a 486 byte file on the receiving phone. The same process measured 23 times.*

When we calculate the local processing bandwidth for the receiver we get that the phone only process around 10 bytes per second. In the current situation 10 bps is enough since it takes an average of 22 seconds for 140 bytes message to arrive [Appendix II]. If the network protocol was able to transfer the file faster in the future we might end up in a situation where the receiver of the file would be the systems bottleneck.

6.1.3 From sender to receiver

From the beginning of the project we expected that the network protocol would be able to transfer the message from sender to receiver in a matter of a few seconds. As it turned out, this is not the case. The test showed that small SMS messages generally are given a higher priority on the network than the bigger ones. This means that the header SMS arrives as the first message in 100% of the test cases. It seems that this is not only because it is the first message to be sent, but also because it contains less data than the following messages. This statement is based on the fact that in more than 50% of the test cases the last message, which is only half the size of the three prior messages, arrives as number two or three even though it was sent as number five. In other words, small messages seem to outpace large messages but not the other way around.

As shown in Appendix II, it takes an average of 5 seconds from the message is delivered in the outbox of the phone until it is passed on to the network. This is a potential bottleneck and one of the sources behind the amount of time it takes for the message get transferred. Another bottleneck is the time it takes for the network protocol to transfer the files. As shown in fig. 9 the time it takes for the last and fifth message to arrive is much more uncertain than the first message. It is easy to imagine how the uncertainty would increase as the number of messages rise to a thousand. This brings in other issues of concern; how long the receiver should wait for messages to arrive before closing the file and should there be a limit on the size of the files allowed to transfer?

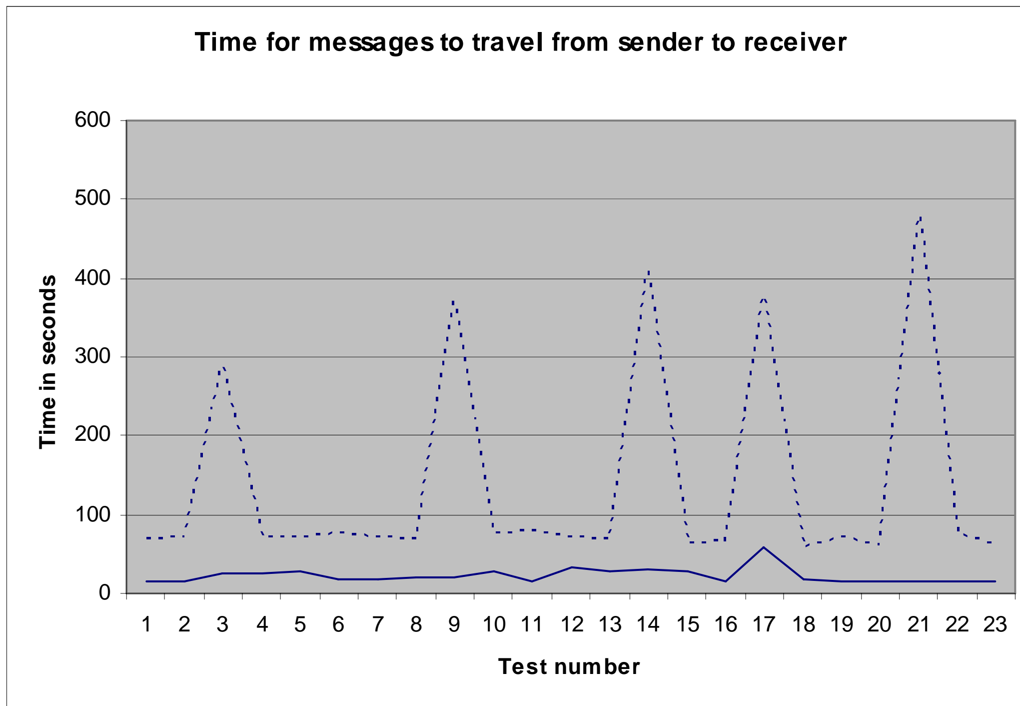

Fig.9. *The time it takes for respectively the first and fifth (dotted graph) message to arrive at the receiver's phone. Measured from the time the sender pushes the send button. The same process measured 23 times.*

6.1.4 Total bandwidth

As mentioned earlier a message containing 140 bytes will take in average 22 seconds to get from the senders outbox to the receiver's inbox. This gives us a total bandwidth for the network and phone processing on *6.36 bytes per second*. If we subtract the time it takes the test phone to deliver the message to the network, namely 6 seconds, wee see that the bandwidth of the network alone is *8.75 bytes per second.*

6.2 Testing on the emulator

As mentioned in the beginning of this chapter, the lack of a testing environment for network communication means that the emulator only can be used to test the parts of the program that is about local processing. As mentioned in the Implementation chapter we tested the full functionality by moving the messages to the *inbox* instead of the *outbox* to simulate an arrival of a new message.

We also made a test of the processing time of a 486 byte file from the sender's view. The test was made to see if there was a pattern similar the test made on real phones. If this was the case the emulator tests could be expanded to get a larger result-set to elaborate from. Fig. 10 shows, by comparing to fig. 7, that the emulator does not emulate the operating system's access to either processor or memory. Therefore, there was not made any further emulator test with the purpose of getting benchmark results.

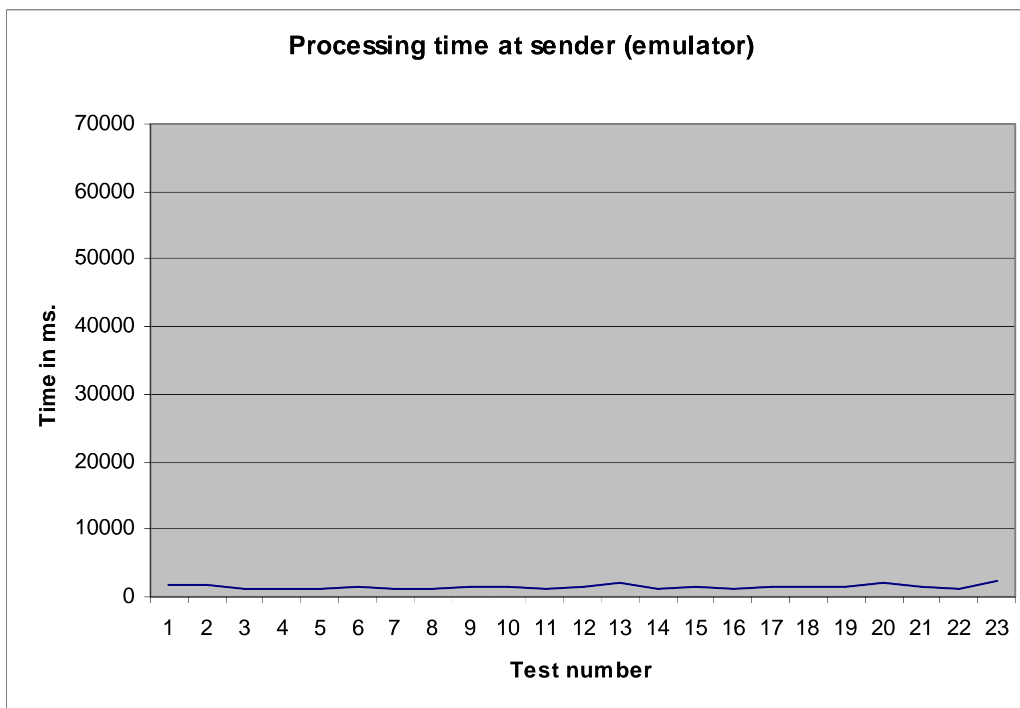

Fig.10. *The time it takes to process a 486 byte file on the emulator from the sender's view.*
The same process measured 23 times.

Fig. 11 shows that the amount of bytes the emulator can process per second is based on the size of the machine running the emulator and not upon the size of the machine emulated.

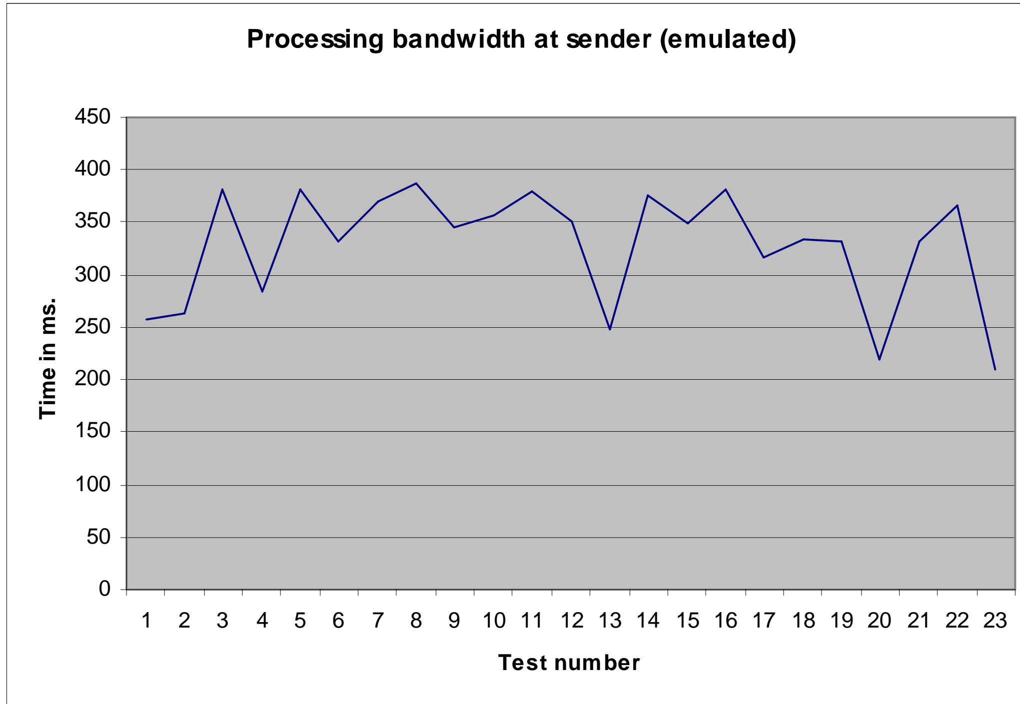

Fig.11. *The calculated bandwidth when processing a 486 byte file on the emulator from the sender's view. The same process measured 23 times.*

7 Conclusion

This chapter will sum up our project. The group will reflect on its experience using the Symbian C++ language as first time developers. We will look at the developed program and reflect on its future, if any. Finally we will review the problems of the problem statement and determine if our goals have been reached.

7.1 The Symbian Experience

Leaning Symbian has not been easy we must admit. We had naively expected our J2ME experience to aid us during development since we were certain that the languages must share a lot of 'wireless' terminology. This was far from the case. We had to start from ground zero which lead to a lot of frustration in the early stages of development. Especially the missing console was the source of much anger. Once the steep learning curve had been climbed and basic principles learned, we found Symbian nice to develop in. The language introduces conventions that we have found to be intuitively appealing. The Symbian SDK furthermore has classes for almost anything. We had not expected build-in classes for heavy load operations like e.g. image conversion and file compression. Very impressive for a wireless platform we think. The Symbian OS has fulfilled our initial expectations regarding low level access. We can do most of the things that J2ME has limited us from in past development and is has been a good experience to learn this at first hand.

The group realizes that we have only dealt with a small part of Symbian C++ SDK in this project. We will therefore refrain from criticizing the language architecture since we do not fell qualified and experienced enough to do so.

'Where do we go from here' we might now ask ourselves. Should we continue developing in Symbian or are other wireless developing platforms the way of the future? We can not answer this question here but only state that since Symbian continues to be the main OS of the world's largest handset producer, being a Symbian developer will stay profitable in some years to come.

7.2 The Product

We have developed a working program prototype. We haven proven our concept via local testing in chap. 5.4. We have managed to port to program to actual phones and do OTA tests. Before commencing with the OTA test we had learned that high bandwidths would not be obtainable via the SMS protocol by researching the expected bandwidth in chap. 3.3.3. Our hopes for bandwidths that would allow limited streaming had therefore already been put to rest. The results that we have collected have due to the 'in-sequence' problem stated in chap. 5.4.1 been few and the validity of these can therefore be questioned. To give a proper image of the SMS protocol bandwidth, empirical data should have been collected and processed to a higher degree. The results that we managed to collect showed a average bandwidth of 6,36 bps.

What could this product then be used for if we were to complete development? As mentioned, we imagine several applications that require low bandwidths, which this project could be applied to. We have already mention synchronization services of calendar and contact system. Another application could be a J2ME game portal that transfers J2ME games to your phone for free. J2ME games usually has a since of 100kb which would take 715 SMS messages to transfer. Despite these few area of application, we acknowledge that our product have limited use. The bandwidth is simply too low.

7.3 Reached goals

To sum of this project we will now hold our reached results against the problems of the Problem Statements. The problems are:

- *Is it possible to transfer files from one phone to another for free or at low cost using the SMS or GPRS protocol?*

We have proven is possible to transfer files using the SMS protocol and thereby answered this question as well as proven our concept. Problems exist that should be dealt with before the product of the project and the service it provides can be considered reliable.

We only managed to find time to work with the SMS protocol and will leave the GPRS protocol to be the subject of future projects.

- *Are the SMS and GPRS protocols reliable and what bandwidths are obtainable?*

The SMS did not deliver in sequence arrival of packages and [14] states that the service will become unreliable in case of heavy traffic load. The SMS is therefore considered unreliable.

- *How should a data file be segmented into chunks to accommodate either of the listed protocols before transmission?*

We successfully designed a way to disassemble a file and by using proper chunk tagging reassemble the file again. Hereby we have answered this question.

- *How big a file can be transferred between 2 phones?*

We did not research this questions based on financial reasons. We have therefore not answered this. Theoretically the only limit of file size is the size of local storage on a handset.

- *Is it possible to stream data between 2 phones?*

Using the SMS protocol proved it impossible to stream data due to low bandwidths. Even width high level compression this would be impossible with the reached bandwidth average of 6,36 bps.

- *How does one insure that all chunks of a segmented file have been received at the destination phone and how should possible lost packages be retransmitted?*

This question was disregarded when we found [1]. Techniques for ensuring chunks arrival and retransmission will be discussed at the oral exam.

- *How should a file be reconstructed from packages at the receiving phone and how should such a file be validated for consistency?*

This question was also disregarded when we found [1]. Validation techniques will also be discussed at the oral exam.

- *Is package encryption an issue that should be regarded and if so, how?*

We did not find time to research neither compression nor encryption schemes as discussed in chap. 3.5.1 and 3.5.2. Doing so would have been straigth forward since the Symbian OS supports both.

Final Words

Despite the unsuccessful result of the OTA testing we consider this project a success. We have answered the main question of the projects and leaned Symbian OS basic along the way. We feel that we have overcome many of the problems we faced during development and that the final product has turned out well.

It is a bit disappointing that our program does not work to its full when testing OTA but we feel that just porting our program to a real phone was an accomplishment. Hope you do to ;-)

8 References

[1] Klaus D. Gradischnig, Stefan Krämer and Michael Tüxen, *Loadsharing – A key to the reliability for SS7-networks*, 2000, http://www.c7.com/ss7/whitepapers/loadsharing_drcn2000.pdf

[2] Ravi Ravishankar, *Next-Gen Signaling Architecture*, December 2002, http://www.tmcnet.com/it/1202/1202pin.htm

[3] Cisco Systems, *Voice Traffic Engineering*, May 2005, http://www.cisco.com/univercd/cc/td/doc/solution/dialvoic/bliss/t1e1/t1e1_dig/dig_ch4.htm#wp1014683

[4] David Crowe, *Cellular Networking Perspectives*, March 2001, IP and SS7: The Signaling Generation Gap, http://www.cnp-wireless.com/ArticleArchive/Wireless%20Review/200103-SS7-IP.htm

[5] Description of the Model-View-Control pattern http://en.wikipedia.org/wiki/MVC

[6] *Java MIDP Application Developer's Guide for Nokia Devices v1.0*, November 2002, Forum Nokia Global Web Site, http://www.forum.nokia.com/info/sw.nokia.com/id/88558599-3451-42ff-9305-3edd7f4ece61/Java_MIDP_App_Dev_Guide_v1_0.pdf.html

[7] *CIMD Interface Specification For Nokia SMS Service Center 8.0*, December 2005, Forum Nokia Global Web Site, http://www.forum.nokia.com/info/sw.nokia.com/id/7a27b9e7-7cdd-4456-b630-3d7c35f30a4f/CIMD_Interface_Specification_SC80.pdf.html

[8] Richard Harrison, *Symbian OS C++ for Mobile Phones vol.1*, 2003, John Wiley & Sons.

[9] Richard Harrison, *Symbian OS C++ for Mobile Phones vol.2*, 2004, John Wiley & Sons.

[10] Peter Sayer, *IDG News Service 11/22/05*, Paris Bureau, http://wireless.itworld.com/4286/051122mobjump/page_1.html

[11] *Essential Symbian OS series: Coding Tips*, 2004, Symbian Software Limited.

[12] *Nokia Connectivity Framework 1.2*, February 2005, Forum Nokia Global Web Site, http://www.forum.nokia.com/main/0,,034-621,00.html

[13] *How to create a SIS file*, November 2003, NewLc website, http://www.newlc.com/article.php3?id_article=207

[14] Technical Information Bullitin 03-2, *SMS over SS7*, December 2003, National Communications System, http://www.ncs.gov/library/tech_bulletins/2003/tib_03-2.pdf

[15] *Tutorial on Signaling System 7 (SS7)*, Performance Technologies, http://www.pt.com/tutorials/ss7_tutorial_091503v2.pdf

[16] *Description of Best-Effort-Delivery* http://en.wikipedia.org/wiki/Best_effort_delivery

[17] *Designing and building portable UI's for Symbian OS*, http://www.symbian.com/developer/techlib/papers/portable_dialogs/Portabledialogsv1.1.pdf

[18] *Symmetric-key algorithm*, http://en.wikipedia.org/wiki/Private-key_cryptography

[19] *Software platform for touch screen based phones running Symbia,* http://www.uiq.com/

[20] *Description of Nokia 6600,* Nokia website, http://www.nokia.dk/phones/6600/

[21] *Description of Nokia 7610,* Nokia website, http://www.nokia.dk/phones/7610/

[22] *Integrating LeaveScan into Code Warrior v.3.0*, http://discussion.forum.nokia.com/forum/archive/index.php/t-60010.html

[23] *Symella*, Budapest University of Technology and Economics, http://symella.aut.bme.hu/

[24] *Skype to hit mobile phones this year*, MobileMag website, http://www.mobilemag.com/content/100/103/C3948/

9 Appendix I Project code

10 Appendix II Test Results

The following contains the numbers gained from the benchmark tests which the figures in the Benchmark chapter if build upon.

Local data processing using Nokia 6600 as sender.

A file consisting of 486 bytes.

1. column: Number of test.

2. column: The time it took from the user pushed the send button until all files were handed to the phones outbox.

3. column: The amount of bytes the phone processed per second.

Test number:	Time in ms:	Bandwidth in bps:
1	6593	73
2	6593	73
3	6015	80
4	6062	80
5	6343	76
6	6906	70
7	7203	67
8	7265	66
9	7046	68
10	6468	75
11	6500	74
12	5750	84
13	6328	76
14	6000	81
15	6968	69
16	8328	58
17	5921	82
18	6843	71
19	7109	68
20	6546	74
21	6578	73
22	6828	71
23	6828	71

Local data processing using Nokia 7210 as receiver.

A file consisting of 486 bytes.

1. column: Number of test.

2. column: The time it took from the user received the message until it was written to file. All files in a process added.

3. column: The amount of bytes the phone processed per second.

Test number	Time in ms:	Bandwidth in bps:
1	50343	9
2	48296	10
3	54890	8
4	54062	8
5	45187	10
6	63515	7
7	35350	13
8	51484	9
9	58328	8
10	48234	10
11	53125	9

Total time span from sender starts process until receiver receives messages.

A file consisting of 486 bytes.

1. column: Number of test.

2. column: The time it took for the first message in a series of five to get to the receiver + average.

3. column: The time it took before the fifth message in a series of five to get to the receiver + average.

Test number:	First sms recieved after: in s:	5 sms received after: in s
1	15	68
2	16	70
3	25	290
4	25	72
5	28	72
6	17	76
7	18	72
8	20	69
9	20	370
10	29	76
11	14	78
12	33	70
13	27	69
14	30	410
15	29	64
16	15	67
17	59	380
18	18	58
19	15	72
20	14	62
21	15	480
22	14	76
23	16	62
Average:	*22.26*	*138.39*

Time it takes for a message placed in the outbox to be passed to the network.

Tested on a Nokia 7210 with messages of 140 byte each.

1. column: Number of test.

2. column: Time before message leaves the outbox.

Test number:	Time in s:
1	7
2	6
3	6
4	5
5	6
6	5
7	7
8	5
9	6
10	6
Average	5,9