Benefits of multithreaded applications

D a n i e l T h u n e l l

Master of Science Thesis
Stockholm, Sweden 2007

ECS/ICT-2007-65

Abstract

This is a master thesis of which the purpose is to find the benefits and increase of performance that are possible when going from a single threaded, sequential application to a multi threaded parallel. The theses will be based on a real life example where a company need an application that will act as a bridge between their existing program and a server to which they need to communicate. The application will be developed in Java and use an existing API for the server communication.
This application will be tested to see how well it interacts with the server, what performance it can deliver and how this performance relates in a multi threaded environment.
We will see that by utilizing the parallelism of a multithreaded application we can boost performance by an incredible 500% compared to sequential computing. This result should be considered a clear indicator of the performance gains made possible from multi threading. Besides sheer performance, writing applications that work with several threads also prepares for future scalability. Giving you the possibility to add more threads at a later point, should the load on the application increase.

Sammanfattning

Det här ett examensarbete vars mål är att hitta de fördelar och prestanda vinster som är möjliga när en applikation går från att vara enkel-trådat och sekventiellt till fler-trådat och parallellt. Examensarbetet kommer att baseras på ett exempel ur verkliga livet där ett företag behöver en applikation som ska fungera som en brygga mellan deras existerande program och en server med vilken de kommunicerar. Applikationen kommer att utvecklas i Java och använda sig av ett befintligt API för server kommunikationen.
Den utvecklade applikationen kommer att testas för att se hur väl den interagerar med servern, vilken prestanda den levererar och hur dess prestanda relaterar till den fler-trådade miljön.
Vi kommer att se att genom att utnyttja parallellismen som uppstår av att använda flera trådar kan vi öka prestanda med hela 500% jämfört med att arbeta sekventiellt. Detta resultat borde anses vara en tydlig indikator av prestandavinsterna som möjliggörs av att använda flera trådar. Förutom ren prestanda ger applikationer skrivna för att använda flera trådar en grund för framtida skalbarhet. Genom att ge möjligheten att lägga till fler trådar vid ett senare tillfälle om arbetsbördan på applikationen skulle öka.

Table of Contents

List of figures

List of tables

1 Introduction with goals and extent

This thesis will handle a problem that commonly arises in the computer application development world. A system built for a specific task that works without complications for that task need to be modified in order to comply with a new situation. The base for this will be a "hands-on", real life example.

A company working with an SMS subscription service has developed and built their own Java™ system. Their system generates the text messages to be delivered to the subscribers of that service, and distributes them to the subscribers' mobile phone carrier. Most of the carriers to which the company deliver text messages use similar delivery systems for injecting the messages into their networks and the companys system is built to handle this. However one carrier has a completely different way of injecting messages. Therefore an additional piece of software needs to be added to the system to cope with that specific way of communicating with the carrier. That additional software is the system that is the subject of this investigation.

This software will act as a bridge between the existing traffic generating platform and the carrier delivering that traffic. In this case the system will distribute the incoming traffic among its threads that send the traffic to the distributor.

This study will have two related but different goals following the development of this system. Firstly how performance can be improved by going from a single-threaded process to a multithreaded process. The other major goal of this study is to highlight some key issues to keep in mind when developing software that is adaptable to new situations. That is, software that can easily be threaded to handle increases in load and software that is easily configurable for handling new situations that arise with changes over time.

2 Syllabus

SMSC		Abbreviation for Short Message Service Center. The gateway to which content providers submit data for delivery to the mobile phone network.
ESME		Abbreviation for External Short Message Entity. The client which connects to the SMSC.
SMPP		Short Message Peer to Peer protocol. Protocol for communication between SMSC and ESME.
Outbound (traffic)		The term outbound will be used for traffic going from the platform to the SMSC.
Inbound (traffic)		Consequently the term inbound will be used for traffic coming from the SMSC and going in to the platform.
Operator / Carrier / Service Provider / Distributor		These terms will be used interchangeable to describe the owner of the SMSC. The party responsible for the delivery of SMS messages to the mobile phones.
Platform		The back-end system to which the bridge will be connected.
Application / Bridge / System		These terms will be used interchangeable for the program that is the subject of this thesis.

3 Description of system

The application that is the subject of investigation in this thesis is built to act as a bridge between an existing platform and a mobile phone carrier. The application will handle traffic both coming from the platform and going to the carrier; from our point of view this will be called outbound traffic. As well as traffic originating from the subscribers and coming from the carrier in to the platform, we will call this traffic inbound.

It is important that the developed application not only fulfills the current needs, but also is able to handle an increased traffic load should the number of subscribers grow.

3.1 The platform

The platform is an existing system used by a company to generate and send SMS text messages to subscribers. It is to this system the Application will be connected. It picks up texts from a database and matches these texts against different keywords. Stored in a second database are subscribers who have chosen certain keywords that they want to watch for their subscription. The platform then matches the texts with the subscribers' keywords and generates SMS messages with the matched texts to be sent to the subscribers.

Also, the platform handles incoming SMS messages coming from the subscribers to start and stop their subscriptions. In its current deployment this platform only supports communication with the mobile phone carrier using the HTTP protocol. Since one of the providers the company works with wants the messages for it's subscribers to be delivered over the SMPP protocol, and the company don't want to redesign its current platform for this, an extra layer of software is required. This software layer will behave against the platform just like its current `send` method and then connect to the carrier using the SMPP protocol and deliver the SMS messages.

3.2 The Carrier

The carrier is a mobile telephone service provider, who has granted the company access to their server working as an SMSC. SMS messages that are to be delivered to carriers customers (the company's subscribers) should be transmitted to the SMSC using the `submit_sm` command of the SMPP protocol. From the same SMSC, SMS messages coming from the subscribers can be retrieved. This is the functionality that is to be provided by the Application.

3.3 The Application

The Application will provide two methods that are to be used for communication with the platform.

One to which the platform can send out the SMS messages it generates. This method is constructed so that the platform can send its messages using this new method with as little modification as possible. The Application will connect to the SMSC using the SMPP protocol and it is on this connection

which the outbound messages will actually be delivered and the inbound SMS messages retrieved.

The other method provided by the Application will handle the inbound messages received from the SMSC. It will call the same method the platform uses to handle incoming messages as the old implementation in the platform does. This means that the platform will not notice that the messages have been received using a new method.

Since the outbound traffic load will be much heavier then the inbound traffic load, which consists solely of “start subscription” and “stop subscription” messages, only the outbound traffic will be handled multithreaded and the inbound traffic will be handled by a single thread.

4 Background material

4.1 The SMPP protocol

> "The Short Message Peer to Peer (SMPP) protocol is an open industry standard messaging protocol designed to simplify integration of data applications with wireless mobile networks ..." [1]

The Short Message Peer to Peer protocol was developed by the Irish company Aldiscon, later bought by Logica who in turn handed it over to the SMS Forum organization [2].

It is used to connect a client called an External Short Message Entity (ESME) to a server called Short Message Service Center (SMSC) using a TCP/IP connection on an arbitrary port. Communication between the ESME and the SMSC is handled using a request-response scheme with predefined Protocol Data Units (PDU's). This communication may be handled in both a synchronous fashion, i.e. each peer waits for a response for the last PDU before sending the next or asynchronous where sending and receiving are handled separately.

4.1.1 SMPP protocol versions

The latest version of the SMPP protocol is 5.0 [3], but versions 3.3 and 3.4 are the most commonly used [2].

Versions 1.0 to 3.3 are the original specification as defined and owned by Logica [4]. The first version of the protocol released by the SMS forum after taking over the protocol from Logica was version 3.4.

Version 4.0 was an independent customization of the protocol for the Japanese market by Logica who still owns this version [4].

This is why the latest version of the protocol specification was named 5.0 instead of 3.5, to clarify that it actually is the most recent version. It includes several changes defined during the lifetime of version 3.4 [4].

4.2 SMPP bind modes

The SMPP protocol supports different ways for the ESME to connect to the SMSC called Bind modes. These modes allows for different operations to be preformed on the established connection. Version 3.3 specifies two different bind modes for the ESME; Receiver and Transmitter. The newer version 3.4 added a third complimentary bind-mode called Tranceiver.

4.2.1 Receiver mode

An ESME bound to a SMSC as a receiver can, as the name states, receive messages from the SMSC. These will typically be SMS messages originating from the customers of the provider owning the SMSC. For each PDU received from the SMSC a response PDU is sent back.

From [4] a typical receiver session is shown in figure 1 below. Where the ESME binds to the SMSC, receives two messages from the SMSC, sends responses to each of them, and then unbinds.

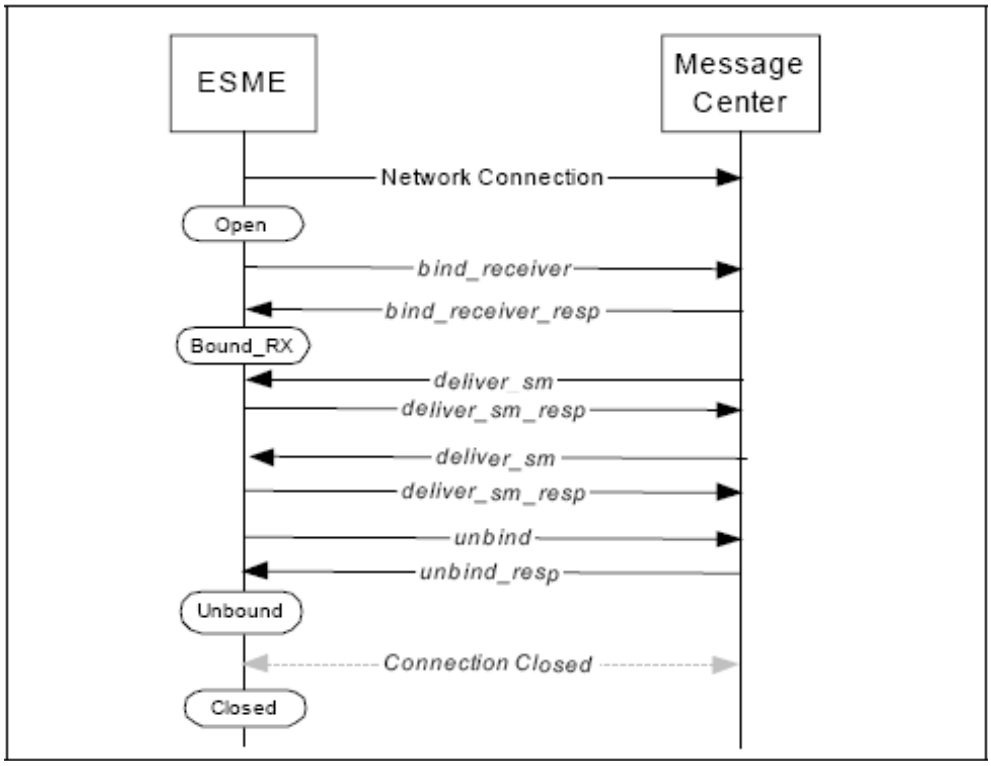

Figure 1: SMPP Receiver Session

4.2.2 Transmitter mode

The transmitter mode is the opposite of the receiver mode. When an ESME is bound as a transmitter it is only allowed to submit SMS messages to the SMSC. These messages are often called mobile terminated messages, i.e. messages whose final destination is a mobile device. For each submitted message a response PDU is received from the SMSC.

Also from [4] is an example session, depicted in figure 2 below. Here the ESME connects to the SMSC and binds as a transmitter. Two messages are submitted and one is cancelled. Each message is followed by a response from the SMSC. Finally the ESME unbinds and closes the connection.

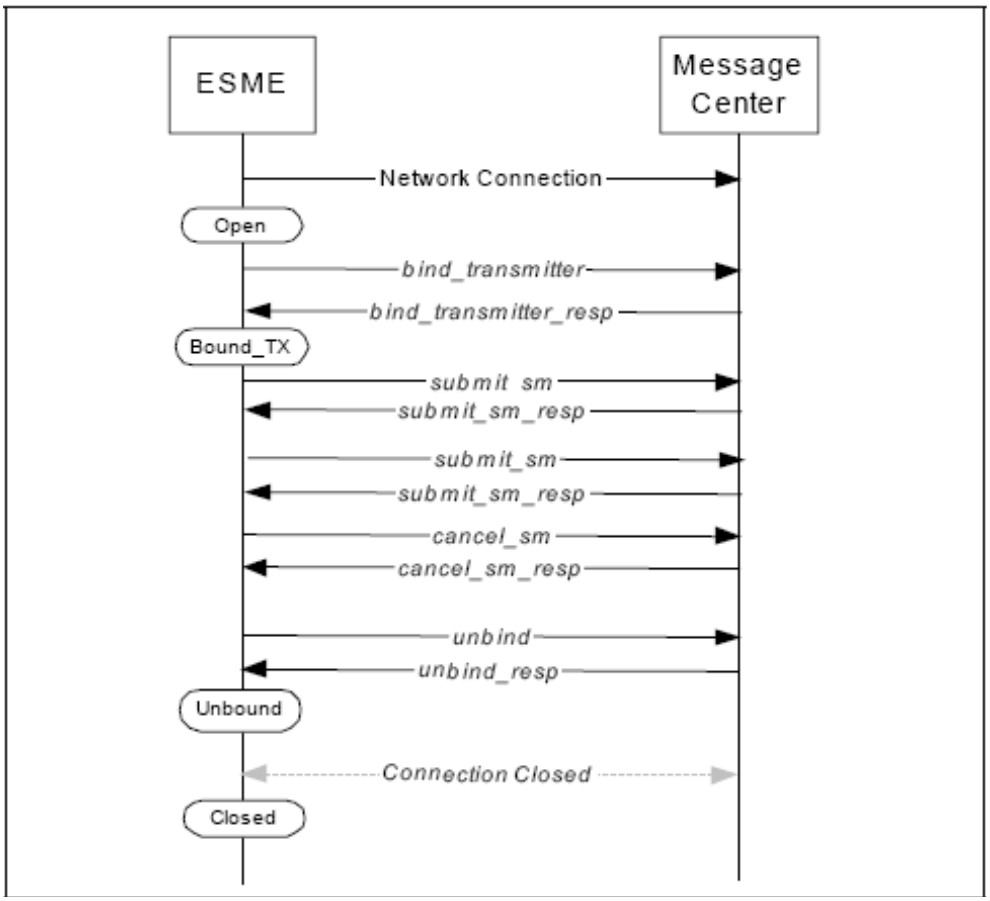

Figure 2: SMPP Transmitter Session

4.2.3 Transceiver mode

Introduced in version 3.4 of the SMPP protocol is the third binding option for the ESME. This is called Tranceiver mode, which as the name suggests is a combination of Transmitter mode and receiver mode. When bound as a Tranceiver the ESME can both send and receive PDU's to/from the SMSC using one single connection. This mode however will not be used in this application.

4.2.4 Synchronization

In the examples shown above all messages exchanged between the SMSC and the ESME are synchronous, i.e. each request is immediately followed by a response. But the SMPP protocol also supports communication in a asynchronous fashion where responses to requests are sent and received in an arbitrary order.

4.3 SMPP API's

There are several different API's available to use for SMPP development. A Google [5] search for "smpp api" yielded in 123 000 results on 2006-01-15. SourceForge [6] hosted 26 project related to SMPP on the same date. Here is a breakdown of the major API's.

4.3.1 The Logica SMPP API

Available from Logicas website [7] is the official API to SMPP. This API was developed by the same company that originated the SMPP protocol itself. It is written in Java and has been made available as open source. It is however no longer supported by Logica as they have handed over responsibility for the SMPP protocol to the SMS Forum group. The latest release of this API is 1.3 which was released on 2001-11-28.

4.3.2 OpenSMPP / SMS Tools

This API is based on the API from Logica and is developed under supervision of the SMS Forum. It is developed as a community effort and released as open source. It can be freely downloaded from SourceForge [8]. Since it is a free, open source project it has no support "help line", but it has a section devoted to it on the SMS Forum's message board [9] where help can be found.

As it has derived from the Logica API it is consequentially developed in Java. The latest version is 2.0, released on 2006-05-09.

4.3.3 Other API's

There are several other API's available, both commercial and open source. Among the open source API's these can be mentioned to show the width of programming languages supported.

- RoaminSMPP [10]. Developed in C#, fully compliant with SMPP version 3.4
- Easy Messaging Gateway [11 http://sourceforge.net/projects/easymessaging]. Written for the .NET 2.0 platform in C#
- OSERL (Open SMPP Erlang Library) [12]. Erlang implementation of the SMPP protocol, covering the entire version 5.0 specification.
- Lightweight PHP SMPP API [13]. PHP implementation of the SMPP 3.3 and SMPP 3.4 API.

There are other API's are available for programming languages such as C, C++ and Python, as well as another Java API [14] independent from the Logica and OpenSMPP API's.

5 Problem formulation

The problem that has been investigated is how to best build an extension to an existing system. The new application should efficiently interact with the existing system without having to change too much in it. As with all applications it should work efficiently and be reasonably tolerant. As it will be a system working in a live communication environment it is important that it is prepared to be easily scalable, so that can be expanded on short notice if the load to be handled increases rapidly. To support this, the developed application will work multi-threaded, to utilize the parallelism of modern computers. This application should have considerable performance improvements compared to ordinary sequential computing.

6 Design

6.1 Programming Language and API

6.1.1 Programming Language

The Application was developed in Java, this since the existing system to which the Application was to be connected was developed in Java. Of course this does not necessarily mean that the Application would have to be implemented in Java as well, but the task of developing the Application came with the preface that it should be. The Java version to be used was 1.4.2 of the Java SDK, this in order to be compliant with the Platform with which the Application will interact and the JVM it runs on.

6.1.2 API

The first choice in the development was to choose which API to build the new application around. For the first cut all the API's developed for programming languages other than Java were removed. When there were several Java API's available there was no need to complicate things by using an API written in another language and write some wrapping code for this.

Since it had to be a free API, a preface made by the company, there were only three choices to be consideration; The Logica API [7], the OpenSMPP API [8] and the Java SMPP API [14]. One major reason for this selection was that they were the most downloaded from SourceForge, and should therefore have undergone enough inspection to be considered reliable code wise and conforming to the protocol specification.

In the end the choice fell on the OpenSMPP API. The Java SMPP API felt inferior to the others since they both originated from the company that developed the SMPP protocol itself. When choosing between the Logica API and the OpenSMPP API the choice was not obvious. The bottom line is that the OpenSMPP API is essentially just an updated version of the Logica API. So the choice really is a choice between what's been thoroughly used and tested versus what's new and improved. Here the latter was chosen, mainly due to the more active support group, which was considered a valuable resource, should something go wrong (and it would be naïve to assume that nothing would).

6.2 Design

6.2.1 Basic concepts

One main though ran through the entire design process, "keep it simple". Since this was going to be a rather small scale system, running on a server that performance wise is similar to an average PC it was felt necessary not to make things more complicated than they had to be just for the sake of neatness.

From the beginning it was stated that the application should be scalable, i.e. be able to handle an increased load without having to so much changes to the application. To be able to change the settings of the application easily

without having to recompile the entire project for each change Java's properties was used. This allows for values to easily be read from a file and inserted into the running program. To utilize this all values for settings that could be subject to change is stored in a configuration (or properties) file. Such values include IP address of the SMSC, account information with passwords etcetera.

To make the Application easily scalable the best solution is to make it multi threaded. Then you could scale its capacity by adding more threads. The number of threads to be used can be entered into the properties file and the system would scale without recompilation.

6.2.2 Traffic and Work Distribution

One important note while designing this application is how the traffic that the application will work with is distributed. First inbound and outbound traffic can be separated.

The inbound traffic is only used to close a subscription. So in the system where the application will be used the outbound traffic is then many times higher than the inbound traffic. No exact data on this is available but the company approximates that the relation between inbound and outbound is at least 1 against 100, but probably higher. Keeping this in mind one thread to handle the inbound traffic should easily suffice. It's clear that it's the sender thread that needs to be quantified, and the number of threads that will be used is to be determined later.

When designing a multithreaded application you need to think about how to distribute the workload. One idea is to spawn a new thread each time a request arrives. But this has both general and specific drawbacks. As stated on [15]

> "One of the disadvantages of the thread-per-request approach is that the overhead of creating a new thread for each request."

Basically this means that if a server that creates new threads for each request, and the number of request are high, it would spend more time and resources creating and destroying the threads than actually processing the requests. [15] continues.

> "Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption."

This should not be a problem for this specific Application, but it is important to keep in mind that you need to have control over how many threads your application creates. Also specifically for this application the above approach would not be optimal due to the nature of request. The messages that are to be sent will arrive in larger bunches with relatively long intervals. This means that with the above approach you would not utilize the parallelism of a multi threaded system well since one thread would take all the messages associated with one request, process them and probably be ready by the time the next bunch of messages arrive. All while the other threads do nothing.

An approach more suited for this application is the so called Bag of Tasks. It works as follows: One thread gathers the tasks to be preformed and stores them in a "bag". A number of worker threads check the "bag" if there are any tasks to be preformed. If there are, the worker takes them from the "bag" and performs them. Gregory A. Andrews states this about the Bag of Tasks approach in [16]

> "The bag-of-tasks paradigm has several useful attributes. First it is quite easy to use. … define the representation for a task, implement the bag, [and] program the code to execute a task …. Second, programs that use a bag of tasks are scalable … merely by varying the number of workers."

This matches well with the statements made for the Application; it should be scalable and simple. For this application load balancing between the threads is not necessary. It is not interesting which thread sends the messages, only that they get sent.

This approach also handles the second problem with the one-thread-per-request approach, namely that of utilizing the parallelism. If the number of messages that arrive in a single request is large, the first worker to check the "bag" does not have to take all of the messages, but can rather take a subset of messages and leave the rest for another worker (or itself should it be done and check the "bag" before any other thread does).

Another thing that is made easy by this approach is the mean to "tweak" the Application to an optimal number of threads. Of course you could spawn threads dynamically if the number of running threads is less than some pre defined threshold. But this would require more complex code. So for the sake of simplicity this approach is better suited.

6.2.3 Main Thread

The application launches with a main thread that in turn creates and starts the receiver thread and all the sender threads. The main thread is then responsible for receiving the SMS messages the platform has generated and sent to the Application. The main thread distributes them among the Application's sender threads that then will send the messages to the carrier.

6.2.4 Sender Threads

Each of the sender threads takes the SMS messages assigned to it by the main thread, establishes a SMPP connection with the carriers SMSC and sends the messages to the SMSC. When the thread is has sent all it's messages, it disconnects from the SMSC and goes back to idle until it gets another list of messages assigned to it by the main thread.

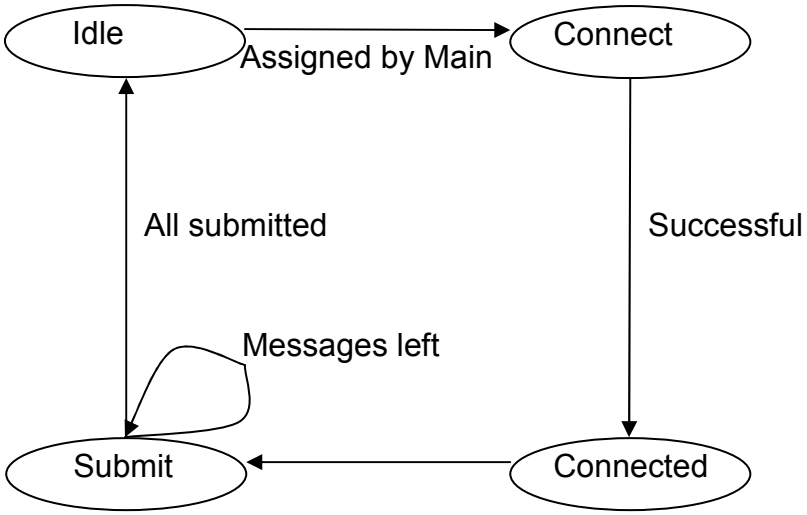

Figure 3: Sender Thread Flowchart

6.2.5 Receiver Threads

The receiver thread periodically establishes a SMPP connection with the carriers SMSC and checks with the SMSC if there are any inbound messages stored. At this point one of two things happens.

1. There are some messages present at the SMSC. These are then retrieved sequentially until none are left. The receiver thread then disconnects from the SMSC and forwards all the SMS messages to the platform.
2. No inbound messages are found at the SMSC at this time. If this is the case the receiver thread simply disconnects.

In both cases, when the receiver thread is done it idles for a pre-defined time before repeating the procedure

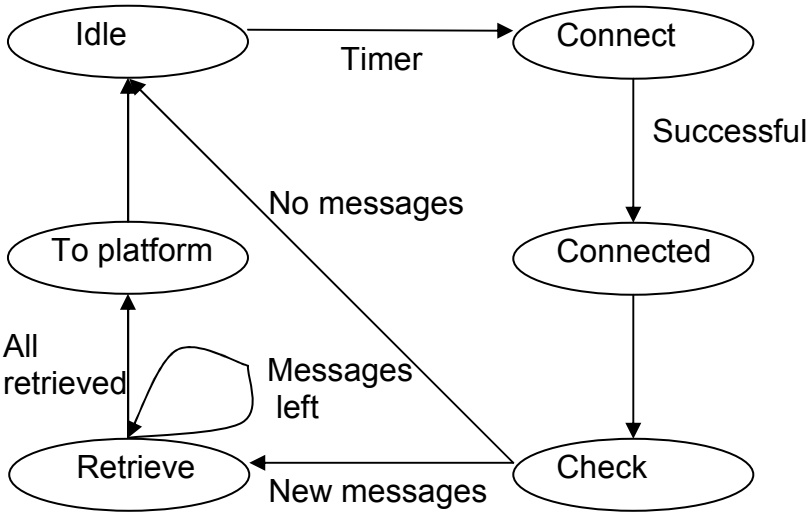

Figure 4: Receiver Thread Flowchart

7 Implementation

7.1 Classes

The desired behaviour of the Application was described in chapter 3. To achieve this behaviour, according to the design from the previous chapter, the Application was developed with one main class, two classes to be threaded. Some additional help-classes were developed as well. This chapter will give a brief explanation of the classes that were developed for the Application.

7.1.1 Server

This is the main-class, i.e. the class that launches the application. It takes the file path to the properties file as argument, creates a `ServerProperties` object to hold these properties and then spawns the `Receiver` thread and all of the `Sender` threads. Then the thread waits until some SMS messages to be delivered are received. These are then distributed in predefined bunches to the sender threads in the order they become available.

7.1.2 ServerProperties

This is the class of which the `Server` class creates an instance to hold the properties. It reads in all the properties from the file and stores them internally. It also provides access methods to these properties so the application can use them.

7.1.3 Receiver

The behaviour of this thread was explained in chapter 3.3.3 and its design is rather straight forward. Create a `Handler` object using the `ServerProperties` object the `Server` thread created and a flag for the bind mode receiver. Check for, and receive, all messages on the SMSC using the `receiveList` method provided by the `Handler`. If the message is not empty, call the `inbound` method of the `MiddleWare` class. When all is done, the thread idles using Java's `sleep` command.

7.1.3 Sender

This threads behaviour was explained in chapter 3.3.2. It starts similar to the receiver thread in that is creates a `Handler` object, but obviously set as transmitter instead of receiver. Then the thread takes the list of outbound SMS messages it was assigned and calls the handlers `submitList` method to send them to the carrier. When it has sent all its messages, the thread will idle until the `Server` thread assigns it with another list of messages.

7.1.5 Handler

The handler class is the class that manages all communication with the SMSC. A handler object is created with the `ServerProperties` object the `Server` thread created. From this all necessary information needed to communicate with the SMSC is available, such as; IP address, port,

username and password. This is the class that accesses the OpenSMPP API. The API provides methods for binding with the SMSC as well as for composing PDU's.

7.1.6 SMS

This class simply stores a SMS message internally. It only has two fields, one for the text in the message and one for the phone number. The phone number field is used as destination number for outbound messages and as source number for inbound messages.

7.1.7 SMSList

The `SMSList` class is used internally for grouping together the SMS objects. It is built around Java's `Arraylist`, but provides its own access methods.

7.1.8 MiddleWare

This is the "glue" between the Application, described in chapter 3.3, and the Platform, described in chapter 3.1. It provides two methods, one that the Platform calls to send messages into the Application and one that the Application calls when it has received SMS messages to send these in to the Platform.

7.1.9 LogFile

This class does not add any functionality to the Application necessary for its operation. It just simplifies the logging in that all classes can use it with just one method call instead of implementing its own logging method.

8 Tests

8.1 Message Generation Speed

8.1.1 The Test

The purpose of this test is to see how fast Application can generate and send SMS messages to the SMSC. To do this a dummy server was set up using the SMPPSim software freely available from Selenium Software Ltd [17]. The version used was 2.2.1. The SMPPSim application mimics the behaviour of a SMSC and is compliant with SMPP protocol specification. When it receives a `submit_sm` PDU it responds close to immediately with a `submit_sm_resp` PDU.

To see how long the application takes to generate and submit a message a timestamp was logged just before every message was submitted to the SMSC/SMPPSim. The difference between these timestamps was then calculated to see how long the application takes to run a cycle of generating and sending messages. For each run 500 messages was sent to the SMSC/SMPPSim and ten runs were made. The result is shown in section 8.1.3.

8.1.2 Test Environment

These are the computers that were used in this test. As client an Acer TravelMate 380 laptop with the following specifications was used.

CPU	Intel Pentium M @ 1.6 GHz
RAM	512 MB
OS	Microsoft Windows XP Professional Service Pack 2 (Build 2600)
Java	SDK 1.4.2.13

Table 1: Client Specifications

As server, a standard desktop computer was used. It had the following specifications.

CPU	AMD Athlon XP 2800+ @ 2,080 GHz
RAM	1024 MB
OS	Microsoft Windows XP Professional Service Pack 2 (Build 2600)
Java	SDK 1.5.0.7

Table 2: Server no. 1 Specifications

The client and server were connected using a TCP/IP connection over a 100MBps Ethernet connection.

8.1.3 Result

Shown in the table below are the values that were calculated from the collected timestamps of each run. Most important is the column that shows the average generation time. This is the time it takes for the application to complete the cycle of generating and sending one message.

Run#	Avg. gen time	Median	Std. Dev.
1	10,4	10,0	6,9
2	8,9	10,0	6,1
3	8,1	10,0	6,2
4	9,3	10,0	14,7
5	8,2	10,0	6,0
6	8,0	10,0	6,3
7	8,5	10,0	6,2
8	8,2	10,0	5,8
9	8,7	10,0	6,4
10	8,2	10,0	6,2
Average	8,7	10,0	7,1

Table 3: Message Generation Test Result

The last row of table three shows the accumulated average of the ten runs. A graphical representation of table three is shown in figure five below.

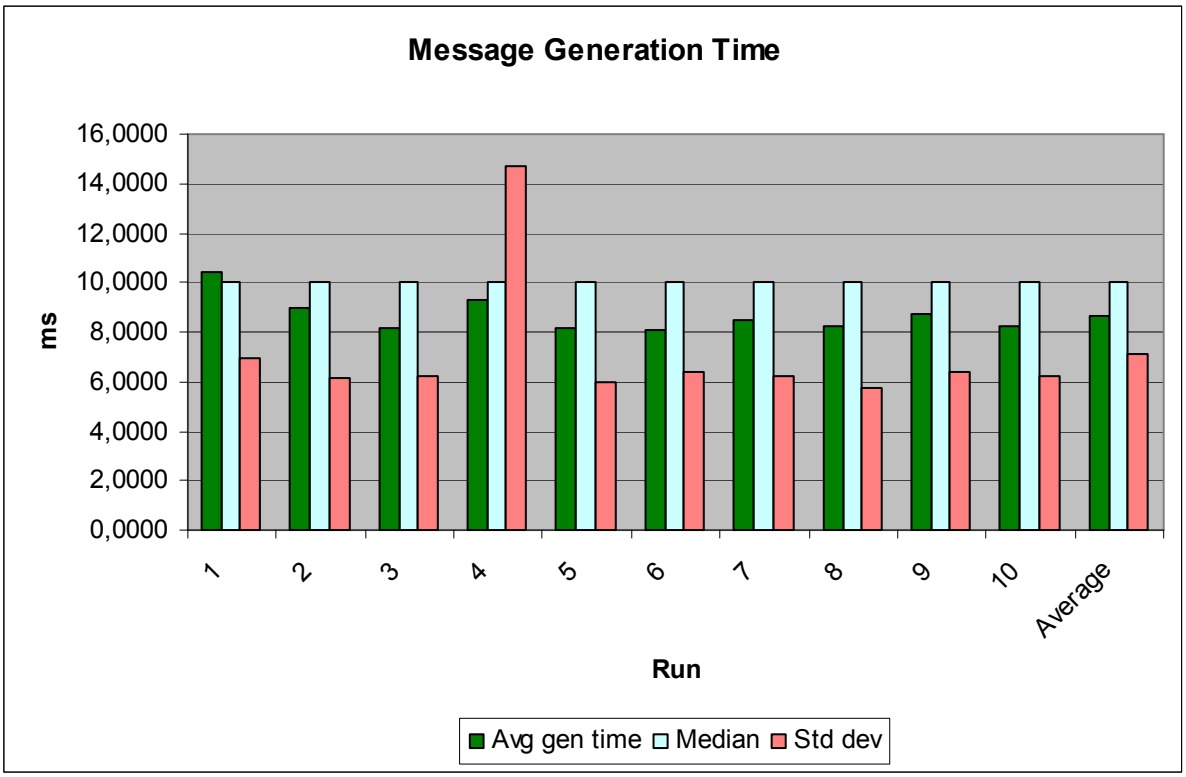

Figure 5: Message Generation Test Result

8.1.4 Conclusion

From table three we see that the average cycle time of the Application is 8,67 ms. This is equivalent to sending 115 messages per second. When the

Application is run on a multi-core/multiprocessor computer each simultaneous running thread should be able to achieve this capacity.

However it is important to notice that the actual number of messages per second achieved is highly dependant on network and server latencies. If one or both of these latencies grow, the number of messages delivered per second will shrink.

8.2 Concurrent message generation

8.2.1 Background

In the original design idea of the Application one of the thoughts with using several threads was that a number of threads would be used in the current deployment to handle the current load and support handling an increased load by adding more threads.

However it turned out that the Carrier to which the Application would communicate with capacity limit of about 5 messages per second. When in talks with the company that were to use the Application it was found out that this was a typical setting, most carriers allowed around five to ten messages per second, while some only allowed one!

8.2.2 The Test

With the Application being able to generate and send one message every 9 ms (as shown in section 8.1) it should be able to send five messages per second to five different SMSC's. If one thread is used for one SMSC each thread should have enough idle time to let the other threads send to their respective SMSC, and this on a machine with a single processor with only one core.

8.2.3 Test Environment

The Application was run on the same client machine described in section 8.1.2, this machine also hosted one simulated SMSC which was reached through its external IP address. The server used in section 8.1 was also used for this test. The other three servers used are described below.

CPU	AMD Duron 1400 @ 1,4 GHz
RAM	512 MB
OS	Mandriva Linux 2007, Kernel 2.6.17
Java	SDK 1.6.0

Table 4: Server no. 2 Specifications

CPU	Intel Celeron @ 1,70 GHz
RAM	768 MB
OS	Ubuntu Linux, Kernel 2.6.15
Java	SDK 1.6.0

Table 5: Server no. 3 Specifications

CPU	Intel Pentium III @ 800 MHz
RAM	512 MB
OS	Ubuntu Linux, Kernel 2.6.15
Java	SDK 1.6.0

Table 6: Server no. 4 Specifications

These three servers were reached remotely using a high speed internet connection. All five servers ran SMPPSim [17] version 2.2.1.

8.2.4 Result

For each run of this test the total number of messages sent by the Application each second was counted. From this the average number of messages sent per second per run was calculated as well as a total average of the entire test. The result is shown in table seven below. When calculating the median only seconds one through eight are considered as the remaining columns have values far lower due to the fact that the list of SMS's to send ran out.

second	1	2	3	4	5	6	7	8	9	10	11	
run												Median
1	32	30	30	31	31	31	33	30	2			31,0
2	31	30	30	29	28	31	29	30	12			30,0
3	15	22	23	23	27	27	26	27	26	26	8	24,5
4	29	31	31	31	30	30	31	30	7			30,5
5	24	25	25	30	32	30	31	31	22			30,0
6	21	22	23	21	23	21	23	24	23	23	26	22,5
7	31	30	28	25	25	21	20	21	21	19	9	25,0
											average	27,6

Table 7: Concurrency Test Result

A graphical representation of the same data is shown in figure six below. Where you can see how many SMS's that the application was able to send in one second during the entire test.

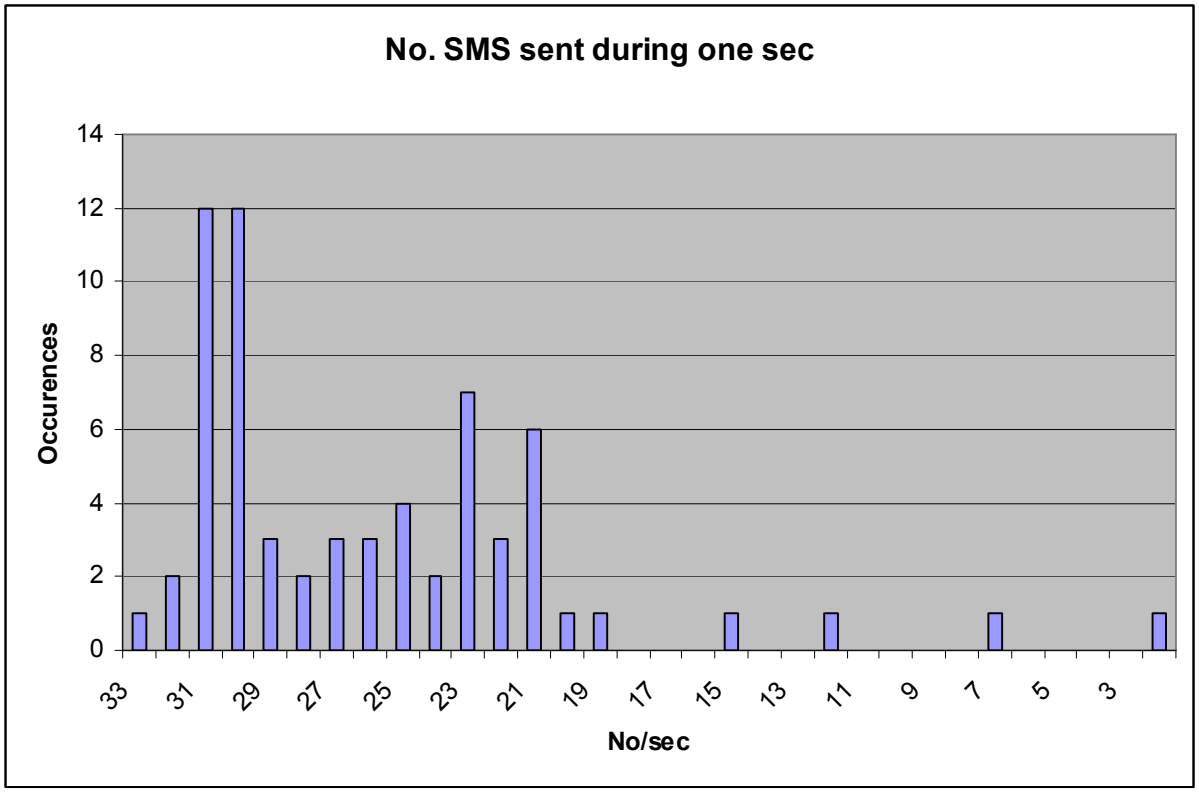

Figure 6: Concurrency Test Result

8.2.5 Conclusion

As shown in the previous chapter the Application can send five messages per second to five different SMSC's, while using only one processor core none the less.

It is notable that in this test the Application still contains some sleep-time on each thread so not to exceed any throttling limit to any of the SMSC's. This means that there is time left for the receiver thread to check the different SMSC's for inbound messages.

9 Presentation and discussion of the results

The tests performed in section eight have shown that there are clear benefits to be made from utilizing the parallelism derived from making an application multithreaded. In this simple example the Application could increase its performance by 500% by distributing its workload on five threads instead of one. This increase should make it apparent that there is a lot of performance to be gained from migrating to a multithreaded environment.

One of the most interesting results from this test is the fact that this increase in performance that came from multithreading was achieved on a computer with only one processor core. This was made possible due to the fact that there were considerable idle times involved that were the result of limitations in communication speed that was set by the Carrier.

If the fact that this enormous increase in performance was possible with one processor core is considered it should be clear that there could be even more performance to be gained in environments in which several processors and/or processors with more than one core are available. In these cases true parallelism can be achieved and you would be able, at least in theory, to increase your performance with the number of processor cores available to you.

10 Conclusions

Servers have long been equipped with several processors and also processors with several cores, but now multi-core processors have taken the leap into commercial computing as well with many, according to [18] Intel expected 70% of new computers to be dual core by the end of 2006. Quad-core processors are just being released to the market [19], and Intel already has produced an experimental architecture with 80 processor cores that they aim to mass-produce in only five years [20].

So in a world where computers with multi-core processors are becoming more and more common, it becomes essential to write code that is either multi threaded to begin with, or is prepared be multi threaded with little or no change to be able to utilize the full potential of these systems. This thesis has shown the gain of performance possible in a client-to-server communication system by utilizing several threads.

However it is important to note that there is a threshold at which point more threads does not add performance due to reasons outside the application itself. This threshold is highly system dependent and could easily change within the same system if different configurations and/or hardware were used. The reasons this happens could be outside the computer running the threaded application such as latencies in the network, or limitations within the computer such as insufficient availability of system memory or simply not enough processing power to utilize the threads. These two problems would cause the system to spend an overhead in switching running threads that would decrease overall performance.

For the system being used in this thesis there was a defined limitation in the Carriers server limiting the capacity of requests that were able to be sent to the SMSC to 5 SMS messages per second. This value could not be exceeded by the client whether multi threaded or not. However, as mentioned earlier this does not mean that this is an absolute top value for the system, just in its current configuration. Adding more server capacity and increasing this limit would leave an opening for more threads at the client side.

Of course there is also a threshold in the client at which point the gain of spawning more client threads levels out. This is because the sheer number of threads is too much for the client system to handle. Again this number increases/decreases with the system configuration.

Due to the fact that the server was limited far below the capacity of the developed Application, the Application was able to use the idle time to communicate with five servers at the same time. This meant that by utilizing the parallel behavior of multi threaded applications performance was increased by 500% in comparison to communicating with all five servers sequentially, and all this on just one processor core.

In general for service systems like this there is definitely an interest in finding a way to distribute the work load on a number of threads for both client and server in order to best utilize the system.

The conclusions above have been derived from these tests that were preformed in this thesis. Which were firstly tests on how fast the

Application could generate messages internally? In which it was found out that the Application far exceeded the limitations set by the Carrier. Secondly a test was conducted to see if the Application really could communicate successfully with five servers at full speed. This test succeeded in transmitting five SMS messages per second to five different SMSC's showing that a huge increase in performance was to be made by going in parallel rather than sequentially.

By developing your applications to be multi threaded you make your program easier to scale in the future should so be needed. You would only need to add more threads to increase your computing power. Even if you don't need it now, you may need it in the future.

Multi threaded applications are especially useful in request driven applications such as this or a web-server when you can let one thread handle each request and thus reducing the overall response time in the system for handling requests.

An important note when writing the threaded code is to do this effectively and think through your design carefully. If for example your program needs to access a shared resource, like a database of a printer, and access to it is sequential. This may, and often will, lead to situations where several threads are forced to wait for other threads to finish with the shared resource, thus eliminating the concurrency gained from multithreading.

11 References

[1] SMS Forum. Message Board, FAQ Section. [Online] http://smsforum.net/smf/index.php?topic=269.0,
[2] Wikimedia Foundation. Wikipedia – The Free Online Dictionary, SMPP [Online] http://en.wikipedia.org/wiki/SMPP.
[3] SMS Forum. [Online] http://www.smsforum.net.
[4] *SMPP Protocol Specification Version 5.0*. 2003. [Online] http://www.smsforum.net.
[5] Google Inc. [Online] http://www.google.com.
[6] Open Source Technology Group Inc. SourceForge. [Online] http://www.sourceforge.net.
[7] Logicas Inc. Logica SMPP API Website. [Online] http://opensmpp.logica.com.
[8] OpenSMPP /SMS Tools. [Online] http://sourceforge.net/projects/smstools/.
[9] SMS Forum. Message Board, Section APIs / OpenSMPP API. [Online] http://smsforum.net/smf/index.php.
[10] RoaminSMPP. [Online] http://sourceforge.net/projects/roaminsmpp.
[11] Easy Messaging Gateway. [Online] http://sourceforge.net/projects/easymessaging.
[12] OSERL (Open SMPP Erlang Library). [Online] http://sourceforge.net/projects/oserl.
[13] Lightweight PHP SMPP API [Online] http://sourceforge.net/projects/phpsmppapi.
[14] Java SMPP API [Online] http://sourceforge.net/projects/smppapi/.
[15] IBM Inc. *Java theory and practice: Thread pools and work queues*. [Online] http://www-128.ibm.com/developerworks/library/j-jtp0730.html
[16] Andrews, Gregory R. 2000. *Foundations of multithreaded parallel and distributed programming*. ISBN 0-201-35752-6. United States of America: Addison Wesley Longman, Inc.
[17] Selenium Software Ltd [Online] http://www.seleniumsoftware.com.
[18] CNet Networks Inc. CNet News [Online] http://www.cnet.com.au/desktops/pcs/0,239029439,240053542,00.htm
[19] Intel Inc. Press Relase [Online] http://www.intel.com/pressroom/archive/releases/20070108comp.htm.
[20] CNet Networks Inc. CNet News [Online] http://news.com.com/Intel+pledges+80+cores+in+five+years/2100-1006_3-6119618.html.