CONCURRENCY IN ANDROID

JANNE PARVIAINEN

Thesis
March 2012
Business Information Systems
Software development
Tampere University of Applied Sciences

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

PARVIAINEN, JANNE:
Concurrency in Android

Opinnäytetyö 50 sivua, liitteitä 3 sivua
Maaliskuu 2012

Haltu Oy on suhteellisen nuori toimija yrityksenä, jonka erikoisalaa on web- ja Android-kehitys. Työtuntien ja aikataulujen kanssa on jokaisessa yrityksessä paineita eikä varaa ylimääräisiin ja odottamattomiin yllätyksiin monesti ole. Tämän takia sovelluskehitysprosessin hiominen mahdollisimman tehokkaaksi on erittäin arvokas voimavara. Haltu nuorena yrityksenä ei myöskään ole vielä kangistunut kaavoihin ja tiettyihin toimintamalleihin, vaan on avoin ja halukas kehittämään omia mallejaan työntekijöiden ja yrityksen eduksi.

Android-kehityksessä täytyy ottaa huomioon tietyt mobiilialustan tuomat erityispiirteet ja vaatimukset, jotka työssä myös käsitellään. Kyseisten vaatimusten sovittaminen asianmukaisen moniajon rakentamiseen tehokkaasti tarjoaa lähtökohdat tälle työlle, joka keskittyy tarjoamaan tietoa relevanteista komponenteista ja tekniikoista ja näin ollen osaltaan tehostamaa koko kehitysprosessia valmiiseen sovellukseen saakka. Keskeisimmät Javan ja Android-kirjaston komponentit käydään läpi sekä niiden toimintaa havainnoillistetaan koodiesimerkein ja pohditaan niiden käyttötarkoituksia yksityiskohtaisesti. Lopuksi esitetään suunnitteluprosessi moniajon toteutuksesta aina vaatimusmäärittelystä toteutukseen ja lisäksi pohditaan mitä näkökohtia on hyvä ottaa huomioon, jotta lopputulos on mahdollisimman tarkoituksenmukainen ja tehokas, tarjoten mahdollisesti koodin uudelleenkäytettävyyttä sekä ylläpitotaakan keventämistä.

Työn tavoite on tutkia ja tuoda esiin syventävää tietoa käytössä olevista komponenteista sekä esittää mallia prosessista, jonka tuloksena on tehokas ja vaatimukset täyttävä moniajo. Tällainen moniajo ei tuota ylimääräistä työtä alati muuttuvien vaatimusten ja ilmenneiden ongelmien johdosta vaan on selkeä ja itsenäinen osa sovelluksen rakennetta. Tarkoituksena on tehostaa sovelluskehitysprosessia ja tarjota näkemystä moniajon rakennukseen, jolloin voidaan tehostaa kehitysprosessia ja säästää siihen käytettäviä resursseja

Käytännön työnä opinnäytetyöhön liittyi turvakamerasovelluksen toteutus, joka mahdollistaa palveluun kirjautumisen, saatavilla olevien kameroiden kuvien sekä lokitietojen tarkastelun. Verkkoliikenne web-rajapinnan kanssa toteutettiin moniajona, jolloin sovelluksen käyttöliittymä pystyi tarjoamaan relevanttia tietoa käyttäjälle sovelluksen ajankohtaisesta toiminnasta.

Asiasanat: moniajo, säikeistys, android

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software development

PARVIAINEN, JANNE:
Concurrency in Android

Bachelor's thesis 50 pages, appendices 3 pages
March 2012

Haltu, the thesis client, is fairly young company specializing in Android and web development. As a young company, Haltu has a natural desire to refine its processes and working culture, since such resources as time and schedules place constant pressure on projects and unexpected factors are an issue the company wishes to avoid. This is the reason why the refinement of software development processes is such a valuable asset for any company. As a fresh and young company, Haltu has not got stuck with old models and procedures but instead is open and willing to develop its functions further to benefit its employees and the company itself.

In Android development, the mobile platform itself presents certain special characteristics which have to be taken into consideration in addition to the standard development princples. Bearing this in mind, this thesis concentrates on developing a design process aiming to build concurrent behavior on Android platform. The essential Java and Android components provided and their uses are explained in detail and the functionalities are illustrated by several code snippets. The result is a design process from the definition of requirements to the actual implementation of the components. There is also discussion of what different factors need to be considered for the behavior to be efficient and able to fulfill its purpose without being unnecessarily complicated and possibly offer code reusability and ease the maintenance efforts in the future.

The goal of this thesis is to research and analyze the available components for concurrency and provide a frame for a design process workflow model. The purpose of this research is to optimize and enhance the whole Android application development process by providing an additional insight into the possibilities of building concurrent behaviour.

As for the practical part, a security camera mobile application was requested where the end-user logs in with a mobile client and is able to view a user specified set of cameras and their latest images with log information and other data. This application supports already existing web application and its services. The network communication with the web service interface was a clear entity for concurrency, leaving the UI available to provide the user with appropriate information and responses.

Key words: concurrency, threading, android

CONTENTS

1 GLOSSARY

Virtual machine	Software interpreting the compiled Java code and providing a virtual environment where the code runs.
Thread	Miniature process enabling concurrent behaviour. Basic term of concurrency discussed in this thesis.
Android	Open Handset Alliances', which members several major manufacturers, operating system based on Linux.
JavaSE	Standard edition of Java programming language. Android libraries extend JavaSE to support Android system.
JSON	Javascript Object Notation. Simple, text-based, format for transferring data using key-value-pairs.

2 INTRODUCTION

Haltu is a software company located in Tampere, specializing in web and Android development at the moment. As I have been working for Haltu during my internship period, I have had the chance to observe professional development of mobile applications and take part in those projects. As expected, the journey from an idea to an actual published application is long and complicated, involving a large number of unknown factors affecting the course of the project. The customer may not be able to express the desired features properly or the features may change and take longer to finish than expected. Any improvement to the application's design process may help to save resources and improve the development team's output in later stages.

The growing number of mobile devices and their improving capabilities continue to feed the demand for more applications. Also, the requirements for those applications are becoming even more challenging while the majority of the device manufacturers see the third party applications available in the markets as an important marketing theme. Mobile application is expected to deliver almost the same level of efficiency as a software running on a desktop computer and be easy to use and highly interactive at the same time. Furthermore, the booming tablet markets emphasize this effect even further by closing the gap between a mobile device and a laptop computer.

Markets are flooded with a vast variety of mobile applications from private developers as well as from different software companies. The simplicity of bringing new applications to the markets is putting high pressure on developers to build quality applications swiftly in order to cope with the demand and competition. This emphasizes the need for a well-rounded design process where the application's class structure is being built as well as possible right from the start in order to save precious resources, including work hours put into development. Understanding customers' needs and building software requiring the minimal amount of refactoring as the project progresses are the key factors in building an application from the ground up. Often functionalities get refined and optimized once the hands-on experience of the first few versions has been tested. In these kind of situations it is important to have appropriate and clear structure to provide those functionalities, making is easier to modify and change

different, clearly encapsulated, entities. As many of the functionalities in modern mobile applications require background processing, these guidelines are crucial when building concurrent guidelines. Hence the workload and costs are kept at the minimum and the response capability to changing requirements is greatly improved.

Concurrency is usually found in any software with even slightly complex functionalities and there are numerous ways to implement this behaviour. When it comes to mobile devices, efficiency is the keyword. As the processing capabilities are not an issue any longer in modern devices, another issue is surfacing, battery life. The drainage on the battery may become significant if multitasking is not planned out well and it can waste computing resources, even if the application's state is paused or even stopped, especially since threads and other concurrent components usually handle quite demanding background tasks, such as GPS managing and networking.

The main objective of this thesis is to research the advantages and requirements of the key components of concurrency on an Android based device and analyze their purpose in-depth in order to present the benefits and shortcomings of different components and approaches. The research is also supported by a constructive part in which Haltu requested a security camera application for Android, where the user logs in and is able to view a specified set of security cameras and their log files and other useful data. This application will implement appropriate concurrent behaviour which is relevant to its purpose and is embedded into the application from the beginning of the development. Other examples will be referenced from various other projects or specially built code snippets.

The purpose of this thesis on the other hand is to ease and enhance the design process by providing an additional insight of the available components helping to choose the appropriate way to implement concurrent behaviour. If the concurrency is not implemented in the most practical way and it is causing trouble later on in the project. This thesis will provide an oversight and some guidelines to help evaluating the best practice to avoid repeating the same mistakes and eventually help save work hours and money put into the project

Concurrency can be implemented in countless different ways depending on the situation, each with their own benefits and shortcomings. This was the main inspiration behind this thesis, to help develop a process where the application's requirements are taken into consideration as early as possible when designing a structure for concurrent behaviour. This way the behaviour can be divided explicitly into different encapsulated entities and components can be designed especially for these entities, providing the required functionality without unnecessary compromises.

Other components implemented in the application will not be discussed any further than what is required in order to explain any implementation aspects of the concurrent components. This thesis concentrates only in Java techniques and native code, C or C++, is being excluded. Basic understanding of Java, Android and object-oriented programming is expected in order to understand the discussed techniques and paradigms.

3 CONCURRENCY

3.1 What is concurrency?

In software development concurrency is the art of executing multiple different functionalities simultaneously. Cesarini and Thompson (2009, 89) defined concurrency as the parallel execution of different functions without affecting each other unless intended to do so. This statement is quite bold, considering the different meanings of parallelism and concurrency. The term concurrency means that the system or application is able to maintain two or more actions in progress at the same time and the system manages the thread scheduling and provides each of these actions computing time based on their priorities and tasks. Parallel execution on the other hand means those actions are actually executed at the same time. This, however, requires multiple processor cores. (Breshears 2009, 3)

The operating system schedules processes for execution with certain parameters, which depend on what kind of scheduling the operating system is based on. The scheduling can be preemptive or cooperative (non-preemptive). The preemptive scheduling is widely used in modern systems, since it is significantly less error-prone and more equal between tasks, also known as processes, because it is provides some computing time to each process. Harold explains this by saying: "A preemptive thread scheduler determines when a thread has had its fair share of CPU time, pauses the thread, and then hands off the control of the CPU to a different thread". (Harold 2004, 132) The amount of time is relational to the tasks' priority but at least this approach guarantees some computing time for other processes instead of concentrating only on high priorities as cooperative scheduling does. This is why preemptive is more reliable and more stable against starvation and other unexpected scenarios since other processes or threads may actually solve and defuse the issue. The non-preemptive scheduling relies mainly on the logic of processes and expects them to pause and provide computing time for other processes as well. This makes it very susceptible to errors since a single poorly designed code block can cause serious problems if the execution is stuck and cannot make any progress, causing the whole application to become halted until the issue is resolved.

The vast majority of modern software implements concurrency of some sort as it is extremely important for the application's overall efficiency and user experience. As a concrete example, an application may communicate with a web service and exchange data in the background while the user is editing a text document or pictures are being downloaded while user is browsing through already existing pictures. Concurrency enables this kind of multitasking by spawning threads in addition to the main thread[1] handling mainly minor tasks and tasks related to the graphical user interface or GUI.

Even though a process running the application may only possess the mandatory single thread to start with, concurrency can be achieved quite simply by spawning and executing own custom-built threads or platform-provided components running in a separate thread. This enables simultaneous operations to be in progress, which is a necessary and vital feature for the application to be able to fulfill its purpose efficiently while delivering satisfying user experience. Furthermore, a constant interference to the user is one of the aspects developers wish to minimize by using concurrency, since such actions can easily lead to unexpected inputs or otherwise downgrade the flow of the application. This is done by assigning tasks, which can be completed without user input, to the background threads to be executed completely hidden from user. This approach keeps the main thread's overhead at bay and its resources can be focused on its main purpose, running the user interface smoothly and keeping it responsive.

The older, more traditional, approach to programming is to execute processes sequentially. This means the user or other functionalities have to wait for the application to finish its current task before being able to preserve the responsive state or to provide computing time for other tasks. Comparing the sequential approach to concurrent, which lets applications execute necessary functions concurrently in the background, it is obvious why concurrent programming has become an industry standard in almost every software. Nowadays the sequential approach is taken only in the most trivial applications due to its limited flexibility and performance.

---

[1] UI thread or main thread is the thread running the application. Every application has at least one thread, the main thread. This is discussed in further detail later in chapter 5.2 Main thread.

True concurrency is yet to come to mobile world as the processor architecture has its limitations, especially when it comes to the low-end devices. Most of the devices in today's markets have only one CPU which is the reason why concurrency in mobile devices is not as powerful as in normal computer systems. In future, as mobile devices with multiprocessor architecture grow in numbers, concurrency will enhance the actual performance of the applications and is able to execute truly in parallel. (Mednieks, Dornin, Blake Meike & Nakamura, 2011, 142)

3.2 Possible issues with concurrency

Each thread runs independently, regardless of the statuses of other threads, unless programming logic dictates to interact or access the same resources as the other threads, such as databases or variables. These interactions may lead to various unexpected events and locks due to logical errors or race conditions between threads if not handled properly. However, there are various ways to prevent these circumstances. The problem is that in many cases these precautions may cause some performance issues and add unnecessary complexity to the structure. This is the reason why careful evaluation and testing of suitable precautions is highly recommended.

One of the most common issues with threading occurs when threads access shared resources. Threads may update these resources while others are accessing the same values, causing inconsistency and possibly leading to a race condition or a deadlock. One scenario of race condition is a situation where two or more threads are using the same resource to perform an operation and saving the results. Whichever result is saved last is the only one preserved, since all other results are written over each time another thread comes to save its results. Basically, the outcome depends on the timing of events and may very well be inconsistent (Figure 1). Naturally this can be avoided by building a protective mechanism for that resource variable to prevent this sort of behaviour. If implemented carelessly this mechanism may cause further issues in a form of deadlock which is a situation where two or more threads block each other from performing the desired operation (Figure 2).

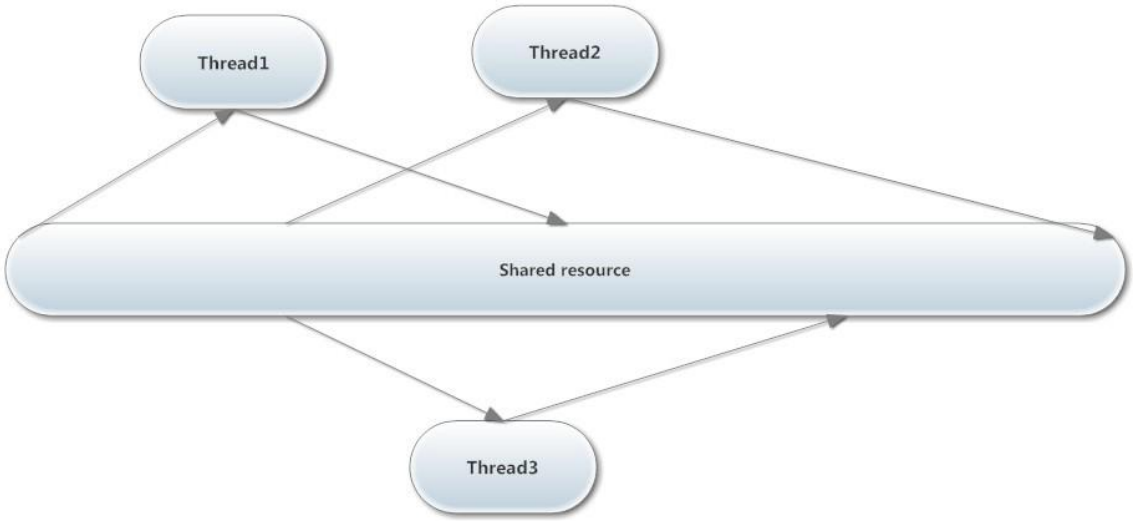

FIGURE 1. Race condition where threads use a shared resource for processing.

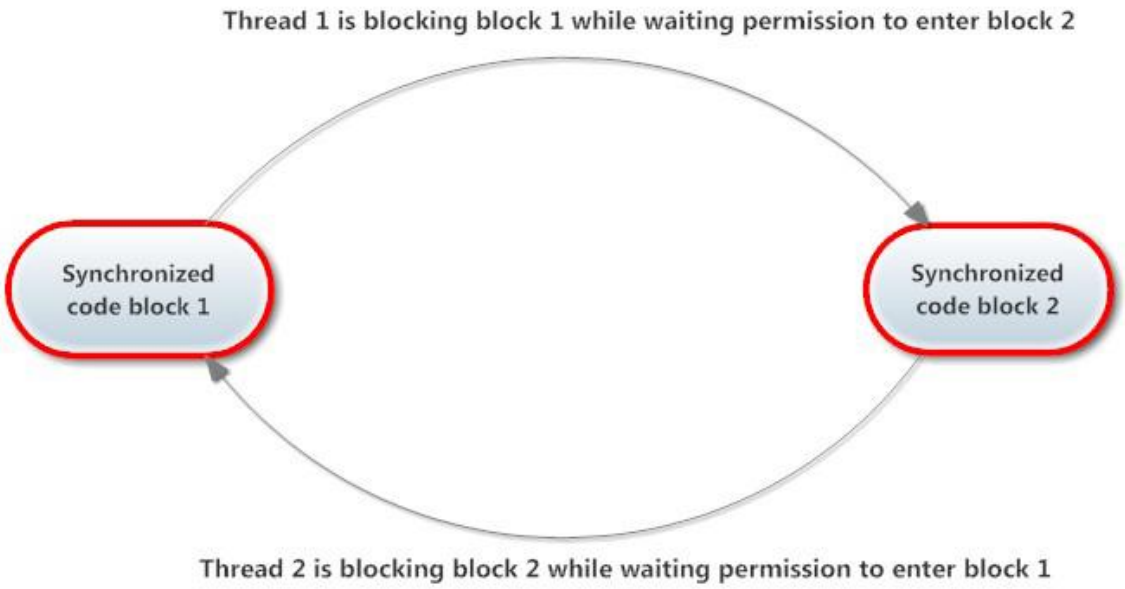

FIGURE 2. Deadlock.

All in all, concurrent behaviour makes the application's structure more complex, requires much more designing and evaluating and is more error-prone than the sequential paradigms. Also performance issues may emerge if some scenario is not handled properly, causing undesired behaviour in the background and consuming resources.

3.3 Threads and processes

Threads and processes are essential parts of any software. It may often seem that these two terms are overlapping, especially since process can have various different meanings depending on the context it presents itself. As for the technical definition in software development in general a process is a container or an environment where the application is executed. It provides the memory space and other necessities needed for the application to run.

So what is the definite difference between a process and a thread? Thread could be defined as sort of miniature process. Threads are located within a process and each process has at least one thread, the main thread. If additional threads are spawned, they also will remain within the application's process as they are dedicated components of that specific application. In conclusion, a thread and a process both are containers of execution but thread is more efficient to create than a completely new process and is simpler to implement. (Processes and Threads, 2012) From programmer's point of view threads are more likely to be used in everyday development.

3.4 Thread-safety

Thread-safety can be defined as a set of measures to avoid unexpected behaviour and data inconsistency with concurrent behaviour and could be characterized as a set of guidelines which lead to certain mechanisms than any actual standardized mechanisms since each application is different and behaves differently. These measures include atomicity, locking measures, liveness and variable isolation among others.

Brian Goetz et al (2010, 22) defined atomic operation in their book Java Concurrency in Practice as follows:

> Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has. An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state.

In the light of this definition it can be said that atomicity is ensuring data consistency by isolating the mutual operation from its operators and its principle is that once it is being executed it will be executed completely. It cannot be interrupted in the middle and any results following from such operation will be invisible until the whole operation is finished. This is usually done by various locks or mutex objects (mutual exclusion) which will be discussed in more detail later on. A non-atomic operation can cause data inconsistency and various conditions where threads are performing undesired behaviour because of race conditions and different locks.

Liveness can be described as timing of component's executions. Loosely similar to what the system is doing with thread scheduling but only on more specific level. How threads time their actions according to other threads or events which define the constraints of the timing of some action. However, a negative aspect with liveness is a situation where the constraints are not changing, hence blocking the thread's access to a block or a resource required for the execution to continue, causing a standstill between two or more threads. Such situation is called a deadlock.

Deadlock is quite hard to identify because it does not manifest itself necessarily. This does not mean that it would not be possible; much of it depends on the thread scheduling done by the system and there are numerous factors affecting this scheduling. There are not any predefined methods to prevent deadlocks as the manifestation is so uncertain and the reasons leading to the lock are changing, depending on the application and its actions and the threading situation at that given moment. Minimizing the use of locks and constraining conditions is minimizing also the chance of deadlock occurring, but this is not always the most practical way and even though the locks have been minimized, the few locks left can still cause a deadlock. It is important to identify the parts in code where there are multiple locks acquired at the same time and check that the locks are not overlapping.

Intrinsic locks are a way to restrict the access to a synchronized guarded block preventing race conditions and deadlocks. Once the thread accessing the resource is finished, it notifies the remaining threads of some event, possibly a changed value in the conditions affecting the guarded block. After this other threads may try to gain access to the guarded block and acquire the lock itself. Intrinsic locks have a major downfall with

deadlock situation. Once the deadlock has occurred, intrinsic lock will put the threads to wait until further notice, in other words indefinitely. A viable option for intrinsic locking is to use a timed lock where the code invalids the lock on behalf of some thread after a specified timeout has passed. (Goetz et all, 2010, 219) However, the problem may still remain due to other deadlock situation with different locks. It has to be carefully analyzed what is the correct manner to handle each locking mechanism.

Semaphores are quite similar to guarded blocks but work as a flag for other threads to act accordingly. After the resource is no longer accessed, the accessing thread releases the lock and notifies other threads. Semaphore can be a counting semaphore or works as a Boolean semaphore. This means that semaphore can grant access to one or more threads where a Boolean semaphore works as a mutex object (mutual exclusion or intrinsic lock), restricting the access to only one thread at any given time. The lock can only be released by the owner of the lock. Semaphores have been implemented into standard Java components since Java 5.

The previous mechanisms, especially guarded block, may very well be quite inefficient when it comes to saving processing resources. If the implementation is non-synchronized the thread waiting for permission to access the guarded block will poll constantly until the permission is granted. This may very well lead to starvation, a situation where a thread is using resources needlessly due to being unable to gain access to the necessary code block. Starvation can however present itself due to various other reasons, such as incorrect thread priorities or infinite loop or failed termination of an instance. (Goetz et all, 2010, 218)

Threads can also get caught in a livelock, a lock where a thread is unable to make progress due to a failure, as demonstrated in figure 3 by the red arrow, or keeps responding to another threads' actions as the gray arrows in figure 3. Adding random timeouts for the components to wait before retrying can solve a livelock if the threads are not able to make progress in any other way.

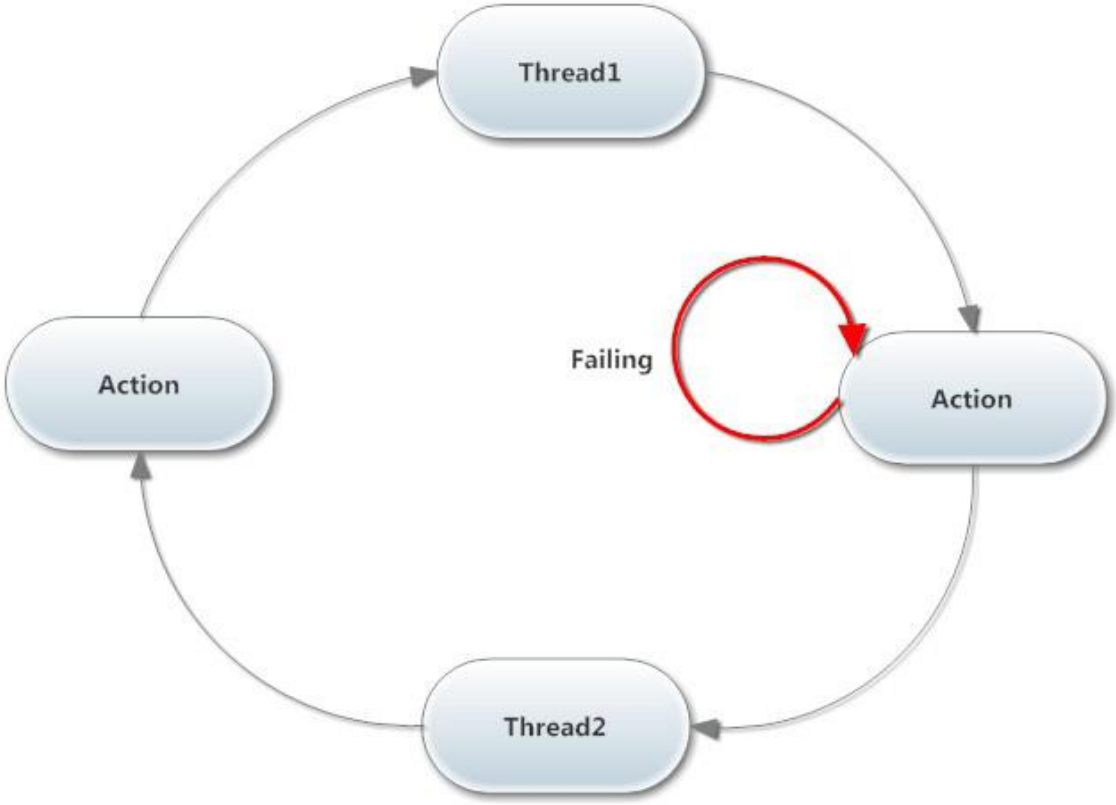

FIGURE 3. Two different manifestations of livelock.

4 THREADING IN JAVA

Java as a programming language as has been used ever since it was published in the mid 90's by Sun Microsystems and has established its place as one of the most popular programming language in modern day's development. It is an object-oriented language based on various other programming languages, such as C and C++. Java is also designed to be platform independent by using virtual machines to run the code and this is one of the main reasons for Java's popularity since it can be run securely on any platform, and add to that the fact that it has a powerful set of components as a standard for implementation and external libraries can be imported to give even more features and possibilities.

4.1 Java standard components

JavaSE provides a wide variety of components for concurrency and parallelism. Often the most effective is the result of the combination of various components; any specific use-case scenarios are impossible to define as the desired behaviour is rarely final and requirements differ between different projects and applications. Some guidelines and instructions for the implementation are, however, possible to draw and be applied to each development project.

4.1.1 Thread-class and Runnable-interface

`Thread` is a standard Java class implementing the `Runnable` interface. It is one of the most common components of concurrency. According to the Java documentation (Class Thread, 2011), a thread is a thread of execution in a program. Meaning, a thread itself is miniature process running concurrently along the side with the main thread of execution. Each application has at least one thread of execution, the main thread, which boots and keeps the application running. Other threads are usually spawned at runtime to execute any background tasks and once fulfilled their purpose, these threads are

rendered idle and the Java's garbage collector kills and removes them, freeing memory for other objects.

In JavaSE, new threads can be spawned either by subclassing the Thread class (Figure 4) or by implementing the Runnable-interface and passing the instance to a thread (Figure 5). The Runnable-interface is implemented by all classes which are intended to be executed concurrently by a thread and the `Thread`-class itself implements the Runnable-interface, therefore the actual secret to concurrency lies within this interface. (Runnable, 2011) The main difference between a class and an interface is that an interface forces a class implementing it also to implement its methods as the interface itself does not have any implementation in its methods, it only declares a set of methods. Basically an interface is only a list of methods which need to be found from the class implementing it. From object-oriented point of view, the recommended way to enable concurrency is to implement the Runnable-interface, as it is quite rare that the actual Thread-class' behaviour need to be modified which would require subclassing it. (Sierra & Bates, 2005, 500) Another valid reason for using the Runnable interface is inheritance. Since Java does not support multi-inheritance, extending a class with no asynchronous behaviour and `Thread` is not possible, thus leaving interfaces and inner classes as only options for concurrency. (Magee & Kramer, 2005, 25)

```
public class Dummy extends Thread{

Dummy thread1 = new Dummy();
thread1.start();
```

FIGURE 4. Thread-class subclassed and instantiated.

```
public class MyRunnable implements Runnable {

MyRunnable runnable = new MyRunnable();
new Thread(runnable).start();
```

FIGURE 5. Runnable class passed to a thread.

As the `Thread` may be implemented as an inner class or as a normal class, as can be seen from appendix 1 depicting the implementation from the security camera application, it can provide local asynchronous behaviour or an asynchronous class which may be instantiated and started from several others classes as needed by calling

`Thread.start()`. Once started the `Thread` will automatically call `Runnable's run()`-method.

Concurrency is also a useful tool when long-running tasks are required, tasks that may not be confined into a single class or may run continuously, hence any component with only an inner class implementation may not be sufficient. (Mednieks et al, 2011, 154) Though it is important to note that thread can only be run once, it cannot be saved into a variable to be executed again at the later stages as the thread dies once it has filled its purpose. The object may however remain in the memory for some time but it will not be able to provide concurrency anymore, just a normal access to its methods. (Sierra & Bates, 2005, 500)

The most complicated aspect of threading is inter-thread communication, in other words how to pass data while maintaining the integrity and consistency without causing performance issues or any other problems. This where synchronization, semaphores and guarded blocks come in, offering building blocks for controlling the access privileges to shared resources..

4.1.2 Callable

`Callable` interface is very similar to `Runnable;` both interfaces are designed to be executed in a parallel thread. However, the most crucial difference between these two is that `Callable` returns a result and is able to throw an exception. Unlike `Runnable`, Callable instance cannot be passed to thread in such manner, which means the paradigm displayed in figure 8 will not work with `Callable`. Instead, `Callables` are usually passed to `ExecutorService`, which basically is a service for handling the execution, termination and tracking of asynchronous tasks. The actual returned results can be obtained from the `FutureTask` the instance was passed (Figure 6).

```
MyCallable myCall = new MyCallable("results will be in uppercase");
FutureTask ft = new FutureTask(myCall);
ExecutorService exeService = Executors.newSingleThreadExecutor();
exeService.submit(ft);
System.out.println("Results: " + ft.get());

Results: RESULTS WILL BE IN UPPERCASE
```

FIGURE 6. A snippet displaying the use of Callable-interface and ExecutorService with relevant prints.

Which one of these interfaces to implement, depends on the use case. If the execution of the class implementing the interface is aiming clearly to produce results, then the `Callable` provides appropriate the means but if the execution, for example, retrieves data from a web service and saves it to database, there will not necessarily be any need for resulting object then the `Runnable` is sufficient.

4.1.3 Timertask

As the `Thread`-class' methods `stop()` and `suspend()` have been deprecated and their use is not recommended, the thread scheduling may not be as straightforward as one would expect. This is why the timing of continuous executions of one thread should be accomplished by the use of `Timertask` instead of looping a regular thread and creating a new instance after a set period of time. `Timertask` is a tool designed explicitly for this kind of looping behaviour, for example polling a web service and retrieving updates. The concurrency of `Timertask` comes from the implementation of the `runnable`-interface, which was discussed in the chapter 4.3.2. Instead of having only `run`()-method, `Timertask` implements also a get-method for scheduled execution time and a cancellation method, which cancels the execution and removes it from the queue. (Timertask, 2011) `Timertask` has the same issue as the `thread` and `runnable`. It lacks a way to communicate with the UI thread by default. Hence a `handler` is needed again to pass the messages back the UI thread if the user interface needs to be updated accordingly.

`Timertask` provides an excellent way of reducing the problems related to continuous looping execution as `wait(), stop()` and `suspend()` –methods will not have be

used in order to control the timing. These methods are very prone to different locks, such as deadlocks, and that is the main reason why `stop()` and `suspend()` among others have been deprecated. (Java Thread Primitive Deprecation, 2011) Other benefit with `Timertask` is that it does not cause to unnecessary overhead for the Java's garbage collector which it may see an idle thread as unnecessary, causing null pointer exception if not handled properly. In conclusion, Timertask is an excellent tool for implementation when a looping behaviour with relatively short looping interval is needed since the underlying thread, with downtime in between executions, is causing minimal overhead to the memory and a constant creation of an instance on every pass through the loop can be avoided.

4.2 Precautionary mechanisms

Precaution mechanisms are quite universal and the spirit and idea remains the same between different programming languages. Language specific built-in components may however affect the need for some mechanisms. JavaSE has a semaphore as a standard component since version 1.5. Semaphores can be implemented in a variety of ways, such as counting semaphore or a Boolean flag. In figure 7 the semaphore has only a single permit to be acquired, hence acting as a lock for the inner code block in a same way as intrinsic lock in figure 8. If the number of permits is increased the semaphore becomes a counting semaphore, enabling multiple entrances into the protected block. Semaphores requires to be wrapped in try-catch-structure and it is essential to take into consideration the fact the if the release of the lock is inside the try-block or outside the structure the lock may not be released, possible blocking the code from other threads indefinitely, therefore finally-block is necessary. Necessary semaphores have to be passed to relevant threads in order for those semaphores to be able to handle the accessibility.

```
private Semaphore semaphore;

public SharedResourceClass(){
    semaphore = new Semaphore(1);
    new Thread(new Dummy(semaphore)).start();
    new Thread(new Dummy2(semaphore)).start();
}

public void doSomething(){
    try {
        semaphore.acquire();
        /*
         * Processing a shared resource
         */
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }finally{
        semaphore.release();
    }
}
```

FIGURE 7. Simple use of semaphore.

```
public void doSomething() {
    synchronized(this){
        /*
         *
         * Protected block
         *
         */
    }
}
```

FIGURE 8. Intrinsic lock.

In figure 9 the code is protected by a guarded block. It puts the current thread to waiting state until it is notified of some event. However, this event may not affect the lock in any way, in which case the thread is put back to waiting state. (Guarded blocks, 2012) If the thread is not put to waiting state, it will loop indefinitely until the conditions are met. This can lead to serious performance issues, especially when the number of threads is high.

```
public synchronized void guardedMethod(){
    while(isLocked){
        try{
            wait();
        }catch(InterruptedException e){}
    }
    isLocked = true;
    doSomething();
    notifyAll();
}
```

FIGURE 9. Guarded block.

Java programming language offers an alternative to isolating variables behind locks. Variable can be declared as volatile. This means that a volatile variable can be modified by multiple threads. Threads are able to communicate in the background when reading or writing a volatile variable. This way the consistency of that variable is preserved. In some cases using a simple variable declaration is more efficient than implementing an actual locking mechanism.

5 THREADING IN ANDROID

Android SDK adds all the necessary tools to Eclipse in addition to Java Development Kit in order to start developing Android applications. For example, there are several new libraries containing new classes for implementation. This variety of classes offers also extra tools for building asynchronous functionality and the advantage is that most of these tools take into consideration many of the known issues in Android development, such as the activity life cycle and communication towards the UI thread, and offer built-in mechanisms to avoid possible issues. However, these components may not always be the best possible solution for the problem. It has to be carefully evaluated which one to choose or to go with some of the JavaSE's standard components and how to combine different components into an efficient and stable structure.

Android uses standard Java compiler to compile the code for the Dalvik virtual machine. This ensures that the Java standard edition is supported almost entirely, not just a smaller portion or a subset, with only minor exceptions, such as `System.print`, which are considered to unnecessary. (Burnette, 2011, 277) When it comes to concurrency, Android supports all the standard components enabling such behaviour.

5.1 Characteristics

The concept of mobile devices sets certain limitations when it comes to hardware, which are necessary to be taken into consideration when developing software for these devices. Android-based devices are a prime example of a fragmented field of various devices. Some high-end models pack enormous computing power, while at the other end there are low-end bulk models, which implement the same operating system, and the users expect the applications to work in these devices just as in those high-end models. This variety of devices is one of the top reasons why Android development is so challenging.

Not all Android devices' characteristics are noteworthy when it comes to concurrency. For example, screen size is not relevant when designing concurrency. However, some of these characteristics may be critical, such as battery life and memory capacity and as this thesis is about concurrency, it will concentrate on the characteristics relevant to the subject. In many cases threads may execute quite heavy operations in the background or a background service may be keeping the object in running state, even if the application itself is not and therefore consume the battery. When it comes to mobile development it is always a compromise between staying in sync and conserving device's resources, although users often want the application to stay constantly in sync without taking into consideration the effects it has on the lifespan of the battery. Concurrency, while being trusted with heavy and demanding operations and the ability to run without user interference, is in key position here. The whole lifespan of such object has to be implied in code in order to get the best possible performance to avoid issues with objects being removed by built-in garbage collector which is expected to destroy and clean the idle objects stored in memory, including threads and runnables. Unless the thread or any other runnable component is rendered idle, it will continue to run, or at least remain, in the background populating memory until interrupted. And if for any reason that component has been left with a heavy but unnecessary task, such as GPS tracking when the map is not visible, the effect on battery may be significant. Although it is possible to create almost any number of concurrent tasks, it is recommended to keep this number as low as possible to avoid the overuse of the device's or the virtual machine's resources.

Android activity's lifecycle (figure 10) and device's orientation changes also need to be taken into consideration as the activity may cause overlapping objects in course of the recreation of the activity. Therefore, to avoid unexpected behaviour, there needs to be thread handling in activity state methods, such as `onResume()` and `onPause()`. According to Steele and To in The Android Developer's Cookbook the `Thread's stop`()-method is deprecated "because it might leave the application in an unpredictable state". They suggest interrupting and nulling the running threads in activity's `onStop`()-method or setting the thread as daemon. (Steele & To, 2010, 57) Declaring a thread daemon means the virtual machine considers the daemon threads to be purely background threads which are to be interrupted and killed immediately if the application's main thread is killed and all the remaining threads are daemons.

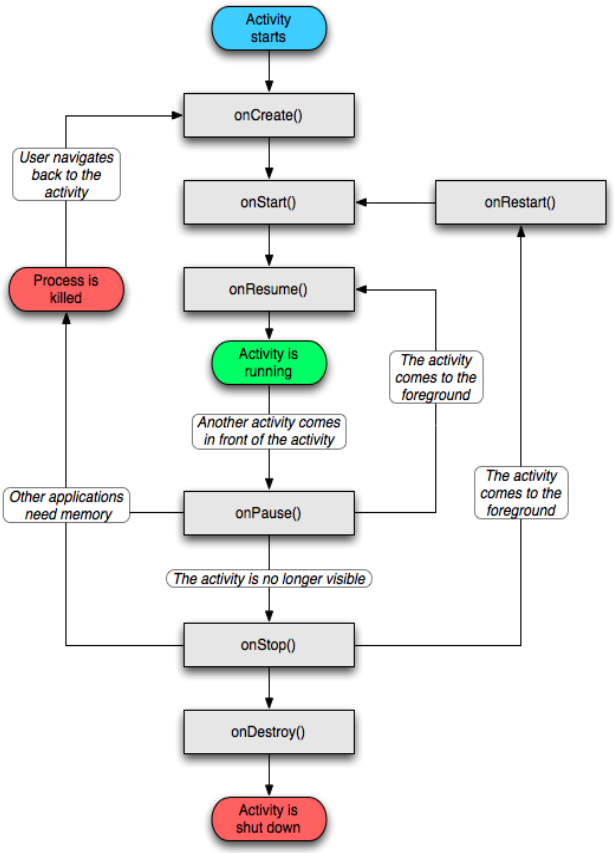

FIGURE 10. Activity lifecycle. (Activities, 2011)

Another issue with mobile devices may be memory capacity. Although memory capacities have grown in recent years, it is still possible to run out of memory with careless coding; besides it is good coding practice to use memory sparingly, especially since mobile devices usually have several applications in their stack in paused state, in addition to the one running. The system is making the call which applications it is keeping in the stack and which ones to kill.

In Android systems each application runs in its own Linux process, which by default contains all the components of that given application. (Processes and Threads, 2011) Process is not the same as the application nor is it the same as the activity within the application. Burnette (2011, 23) explains this as follows; “In Android, an application can be ‘alive’ even if its process has been killed. Put another way, the activity life cycle is not tied to the process life cycle. Processes are just disposable containers for activities.”

System itself tries to uphold the processes as long as it can, but at some point the old processes need to be removed in order to maintain sufficient memory resources. This removal is based on the importance of each process which is determined by the states of their components. There are five ladders in this importance hierarchy: 1. foreground process; 2. visible process; 3.service process; 4. background process; and 5. empty process. The most important process is the foreground process and the empty process has the lowest importance. (Processes and Threads, 2011)

Activity's lifecycle (Figure 10) in Android demonstrates how activities loop from method to method, depending on their state. This paradigm is extremely useful but there is a risk of creating unnecessary duplicate objects, for example threads created in `onResume()`, especially when orientation change occurs since the activity is then recreated and the method stack executed all over from the top.

5.2 Main thread

The UI thread or main thread, if there is not graphical user interface, can be found from every application running on any platform. It is created by the system when the application is launched. (Painless threading, 2011) This is the thread the process holding the application is using to actually run the application. Although most of the code is executed in the main thread its main purpose generally consists of building and upholding the graphical user interface and publishing updates to it. Most of its tasks are relatively trivial and only lasts a short period of time.

If the main thread is blocked for several seconds, Android dispatches an alert dialog for the user informing that the application may not be responding and presenting an option to force close the application, even though the application may be running normally but the main thread is just handling some long-running task. Such a dialog will not reflect positively on user-experience and what kind of impression the application is leaving.

In any application there are few guidelines to assigning tasks for the main thread; 1. Do not block the UI thread, 2. Android UI toolkit can only be accessed from the UI thread. (Painless threading, 2011) Therefore, if application is implementing some long-running

or otherwise demanding task, it should be offloaded to a different thread and only its results imported into the main thread if needed.

There are several ways to communicate towards the UI thread in Android. One is to pass an instance of the activity to the thread, giving an access to `Activity.runOnUiThread(Runnable)`-method (figure 11), which means the runnable given in parameters is executed by the UI thread and therefore this runnable may also publish results on the graphical user interface. Downside is that if the runnable is executing some demanding task it can block the UI thread, even though it is an additional thread. Problem is that this thread is executed by the UI thread. One solution is to spawn a thread doing the actual processing after which another runnable is spawned but its duty is only to publish the results on UI thread. This way the effort required by the UI thread is minimalized. The same logic is valid the View's `View.post(Runnable)` and `View.postInvalidate(Runnable, long)`-methods (figure 11). The downside is also the fact that if the task relatively demanding the UI thread is again blocked or an additional publishing thread is needed between the workerthread and the UI thread. This, however, adds futile complexity to the class structure and logic.

```
activity.runOnUiThread(this);
activity.runOnUiThread(new ClassImplementingRunnable());

view.post(this);
view.post(new ClassImplementingRunnable());
```

FIGURE 11. Tasks executed on UI thread.

Android provides help for this issue, special classes which are designed to reduce the complexity and provide a simple gateway between threads. Most important auxiliary classes are the `Asynctask` and the `Handler`. Asynctask is a complete component which can be used instead of `Thread` and `Runnable`, and it is discussed in further detail in the next chapter. `Handler` on the other hand is a tool which is used with `Threads` and `Runnables`. Handler is used to send and process `Messages`, a class for delivering data values or objects to `Handler` (Message, 2012), which are able to carry information between threads. The Handler is bound to the which created it, and therefore holds is implementation, and adds sent Messages to that thread's message queue. (Handler, 2012) This way the UI thread does not have to run any `Runnables`

in order to publish results, Handler just receives them and acts accordingly. The actual implementation of `Handler` is not within the thread, it is always in the receiving end and only an instance of `Handler` is passed to the thread, in the case from figure 12 the variable named handler is passed to the thread from which we want the data to origin to the UI thread. One example of how to create a `Message` is shown in `getFootage`-method in the appendix 1.

```
public abstract class BaseActivity extends Activity {

    public static int progressDialogSwitch = 0;
    protected ProgressDialog progress;

    protected Handler handler = new Handler(){

        @Override
        public void handleMessage(Message msg){
            if(msg.what == Consts.NETWORK_ERROR){
                Toast toast = Toast.makeText(
                        BaseActivity.this,
                        BaseActivity.this.getString(R.string.network_error), 3);
                toast.show();
                if(progress != null)
                    progress.dismiss();
            }else{
                if(BaseActivity.this != null)
                    onServiceCallback(msg);
            }
        }
    };

    protected abstract void onServiceCallback(Message msg);
```

FIGURE 12. Implementation of Handler

5.3 Asynctask

Android imports one outstanding class in its libraries for handling background threads. This tool is `Asynctask` and it is the recommended tool for executing background tasks concurrently. (Painless threading, 2011) `Asynctask` can be used from API level 3 up, which means that the system itself needs to be in version 1.5 or higher.

`Asynctask` is an Android system provided Java class designed to enable easy implementation of asynchronous tasks, hiding many of the threading details. (Mednieks et all, 2011, 143) `Asynctask` is designed in a way which requires subclassing as the class itself is abstract. The instance of `Asynctask` is created on the UI thread and provides an easy access for updating the user interface without having to deal with

different threads and handlers in order to publish results. This is achieved by encapsulating the component into methods from which some of them are running on background thread. (Steele & To, 2010, 209)

`Asynctask` creates a background thread, so called daemon thread, to run its `doItBackground(datatype…values)`-method concurrently. After this method is executed, `publishProgress(datatype…values)` is used to invoke `onProgressUpdate(datatype…values)`, which is used to publish results of the processing progress on the UI thread. After finishing its tasks, `Asynctask` calls `onPostExecute(datatype…values)` where final codes are executed before finishing. All the other methods are run in the thread which created the instance except for the `doItBackground(datatype…values)`. Not only is `Asynctask` thread-safe, it is also type-safe, as the types of the final and progress results are defined in the parameters of the actual subclass of `Asynctask`. (Mednieks et al, 2011, 148)

`Asynctask` also provides a cancellation method since the expected parameter passed for processing is usually an array or other type of iterable. This way it is possible to stop the execution from any thread after each pass through the `doItBackground(datatype…values)`. It is vital that `Asynctask` has this feature as it takes sets of objects which it processes on one execution, therefore handling relatively long-running operations.

Mednieks et al (2011, 149) point out an extremely important detail about `Asynctask`. Each instance can only be executed once. Calling an execution the second on a single instance will cause an `illegalStateException`. Therefore, every execution of `Asynctask` requires a new instance, hence it is rarely saved into a variable as it is unnecessary.

As `Asynctask` is subclassed and it possesses quite strict constraints in its structure, any non-thread-safe operations may be conducted quite easily. In the case of being implemented as an inner class, it could access the outer class' variables from the `doItBackground(datatype…values)`. This is not thread-safe practice as the method itself is run in a different thread and may lead to inconsistent data values. Therefore, it is not recommended action to take. Instead required values can be passed

as instance parameters and accessed in this manner. However, it is advised that only immutable objects, such as strings and integer or any other objects which cannot be changed, are passed to `Asynctask`. Although, if a mutable object is passed to it should be made sure that only `Asynctask` is holding a reference to that given object. (Mednieks et al, 2011, 151)

In the figure 13 `Asynctask's` lifecycle is demonstrated with a simple string processing example. The execution is commenced by simple instantiating the actual class and passing it with a set of parameters, in this case strings, as follows; `new At().execute("FO", "OB", "AR");`. Strings are switched to lowercase and concatenated into a single string which is returned as a result. Progress can be seen from the logcat prints, located below the actual implementation.

```
public class At extends AsyncTask<String, Integer, String> {

    @Override
    protected String doInBackground(String... params) {
        String temp ="";
        for(int i=0; i<params.length;i++){
            Log.d("doInBackground", "Processing: " + params[i]);
            temp = temp + params[i].toLowerCase();
            publishProgress((int) ((i / (float) params.length) * 100));
        }
        return temp;
    }

    protected void onProgressUpdate(Integer... progress) {
        Log.d("onProgressUpdate", " " + progress[0] + "%");
    }

    protected void onPostExecute(String result) {
        Log.d("onPostExecute", "Processed: " + result);
    }
}
```

```
12-19 21:03:38.343    1   65   ActivityManager      Displayed activity parvi
12-19 21:03:38.423    D   366  doInBackground       Processing: FO
12-19 21:03:38.945    D   366  onProgressUpdate      0%
12-19 21:03:38.945    D   366  doInBackground       Processing: OB
12-19 21:03:38.983    D   366  onProgressUpdate      33%
12-19 21:03:38.993    D   366  doInBackground       Processing: AR
12-19 21:03:39.023    D   366  onProgressUpdate      66%
12-19 21:03:39.063    D   366  onPostExecute        Processed: foobar
```

FIGURE 13. Asynctask's lifecycle.

5.4 Service

One of the distinguishing features in Android is its ability to execute programs in the background as services. (Burnette, 2011, 241) Good examples of such situation are a music player continuing to play music even though the user switches over to other

applications or an email application keeps checking for new messages. If such feature is required, possibly started on device boot, the concurrent class providing this behaviour could be encapsulated into a `Service`. `Service` is an Android library class designed for long-running operations which do not require a user interface or interference and activities can bind to it in order to establish a persistent connection. (Services, 2011) `Service` itself is not a thread nor is it running on separate thread by default. Concurrency has to be implemented by the content of the service, such as `Asynctask` or `Runnable`. Every implemented service class has to be extended from the abstract `Service`-class or its subclasses. (Steele & To, 2010, 65)

One of the most important benefits with `Service` is that tasks inside it are not dependent on activities or their states, therefore `Service` is able to continue executing even after the application is put to background, making it useful tool for concurrency in some occasions. "Tasks that are meaningful to continue even after the component stops should be done by launching a service. This ensures the operating system is aware active work is still being done by the process" as Steele and To (2010, 65) describe in their book on the subject of when and why to use a service. As the Android system reclaims system resources when needed, reclaiming resources from an instance of `Service` is unlikely. (Mednieks et al, 2011, 79)

`Service` could be characterized to be somewhat similar to `Activity` when it comes to different states and therefore offers built-in methods to control its lifecycle, as can be observed from figure 14. This ensures that the service can be stopped and restarted if deemed necessary. (Mednieks et al, 2011, 79) `Service`'s lifecycle does not necessarily go along with the application's lifecycle since they may be started at different times. For example, if application has required permissions and the `Service` is declared to start on device boot in AndroidManifest.xml (figure 15), it will be running long before the application is ever started. Service's own lifecycle also isolates the Service from Activity's lifecycle event, such as orientation changes. This way the state of the objects in the Service, for example a running thread, are preserved and no progress is lost nor duplicate tasks created.

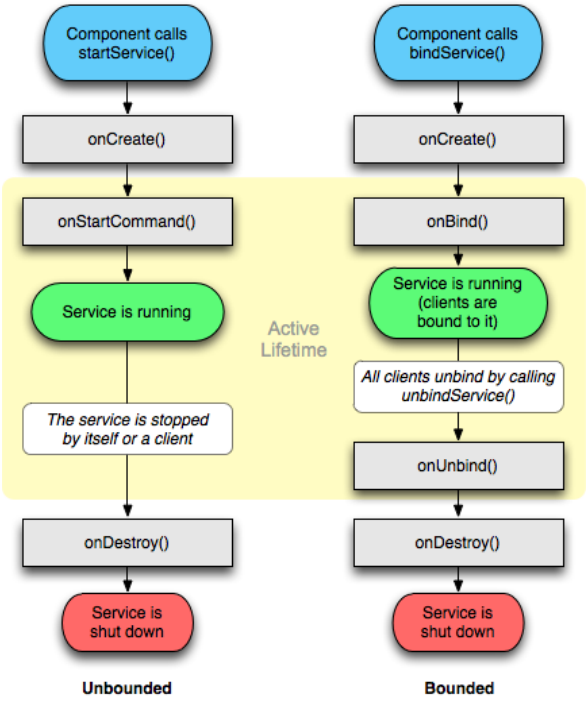

FIGURE 14. Service lifecycle (Services, 2011)

```
<receiver android:name=".service.MessageCheckReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>

<service android:name=".service.MessageCheckService">
    <intent-filter >
        <action android:name="MessageCheckService" />
    </intent-filter>
</service>
```

FIGURE 15. Declaring a service and a receiver class, extending a broadcastReceiver[1], to start on device boot. This action requires a RECEIVE_BOOT_COMPLETED-permission.

[1] BroadcastReceiver is an Android provided class for receiving sent intents either locally or across applications. (BroadcastReceiver, 2011)

6 DEFINITION AND DESIGN PROCESS

Every application's development process starts with definition of requirements. In other words, defining what is the purpose of this application and what it is supposed to do. This is one the most important phases of software development and also one of the most overlooked. Adding or changing features afterwards can cause unnecessary issues as the application's structure may not be able to accommodate them or some unorthodox solutions have to be built in order to solve the issues. Neither one of these options is desired. After the requirements have been defined and the application's purpose is clear, it is time to start outlining the structure of the application. At this point different components play a critical role as how they function together.

6.1 Identifying concurrency

Once application's features have been laid out, separate entities will start to emerge, such as networking, image processing and database communication. These are the parts where you might benefit from concurrent behaviour. If changing the order of steps in a task does not affect the output, performing such tasks concurrently should be considered. (Concurrency and Application Design, 2011) Usually tasks which are running for a long period of time and require relatively much computing resources and do not affect the UI too often should be encapsulated into background tasks and leave the UI thread available to interact with the user while the tasks are being executed.

Each task identified as concurrent behaviour can be broken down to small and compact entities and the actual classes designed to accompany these specific needs. This approach provides flexibility and the class structure is easily expanded to accommodate possible future features and also, the code readability is preserved since the classes executing asynchronous tasks have a clear purpose and they are able to provide a natural gateway to a specific functionality. This adds much to code reusability as the classes remain generic because the functionality is well encapsulated.

6.2 Defining requirements

Concurrent behaviour will be at its best when the behaviour is clearly divided into separate entities and components designed are designed around those entities. However, certain individual requirements must be taken into consideration, especially with Android applications where concurrency components must work together with activities, which have their own lifecycle and states and they are not able to access each other's variables directly.

Other important detail to notice is whether the component, i.e. a `Timertask`, needs to be executing constantly, even if the application itself is not running. Polling a web service and dispatching notifications might be a good example of this kind of behaviour. With such functionality, implementing a `Service`[1] is almost mandatory.

Inter-activity communication must also be spoken for. Is a task started in a different activity than where its results are published? If so, maybe activity inheritance should be considered or at least handlers receiving the results have to be passed and implemented properly in order for the correct activity to be able to receive the processed data. Problematic with this scenario is that the concurrent thread may have pointers pointing to class, i.e. activity, or class variables in an activity which is in paused or stopped state. So to ensure thread-safety, it needs to be carefully evaluated which parameters to pass for the concurrent component as the garbage collector may see objects idle if the activity's state is, for example, stopped. Accessing collected object can cause a `NullPointerException` which needs to be handled or the whole application may crash.

[1] "A Service is an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use" (Service, 2011) This class will be discussed in further details in chapter 6.3.3.

In conclusion, once the application's requirements have been defined, it is time to start drafting the structure of the actual application. Identifying the needs for each requirement is at key position here. For example, displaying data from the web requires a network access class. From these needs, extract the ones which would benefit from concurrency, and possibly from parallelism, and consider how they are needed throughout the application and activity lifecycle and are they accessed from multiple activities. Only after then the design process of the actual component should be commenced. First identify the need and its requirements and fit the components to fulfill that specific need. That way the efficiency is maximized and code is encapsulated properly to ensure the code reusability.

6.3 Designing components

After entities in need of concurrent behaviour have been identified, it is time to start evaluating the actual class structure and which classes to use in order to achieve this behaviour. Almost any class able to run in a separate thread is able to provide the desired concurrency but the key factor is how these classes fit into Android application's principles and its characteristics and do these classes offer reusability or are they built for a single purpose and is executed from the same block of code each time. These are the factors where the right choice of class can make a difference and make the application more efficient and ease the maintenance efforts. These key decisions are shown in the figure 16 which also shows the usual course of designing a component.

In addition to the classes used, standard software design principles help in building stable and efficient structures, such principles include the use of being inner classes, interfaces and inheritance, for example. Together with appropriate classes the concurrency will bring the best of it into the application and give the user the best user experience possible as the application is running smoothly and efficiently.

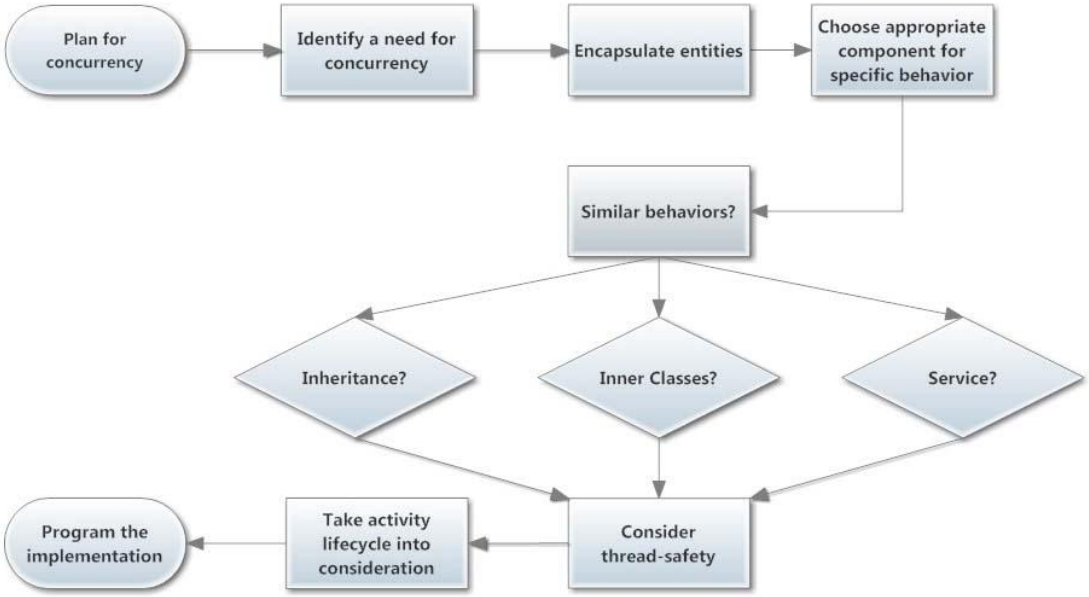

FIGURE 16. Process diagram of component's design process.

Android applications are composed from one or more activities, unless it is only a service without the need for graphical user interface, and the activity's state is changing constantly, depending on what kind of inputs are coming from the user or how the system itself sees the activity, meaning is the system considering the activity to be idle and unnecessary and waiting to be removed from the activity stack. In such occasion if some concurrent class's instance is saved into a variable it might possess pointers to that given activity or its variables and this may cause unexpected issues if that activity is killed and its memory space removed.

After the actual implementation process of the class has commenced, according to Goetz et al (2010) three basic elements should be included; identification of the variables that form the object's state, invariants that constrain the state variables, and establishing a policy for managing the access to the object's current state. Meaning what type of variables is passed to the object, how they are passed and how are they accessed and are there some restraints to the object's behaviour. Similarly it needs to take into consideration how the class is supposed to communicate with other threads, including the UI thread, and if shared resources are used are they confined inside a locking mechanism or are they declared as final, thus preventing any changes to be made, or volatile, variable declaration designed for shared variables between threads.

6.3.1 Inheritance

JavaSE used in Android does not support any kind of multi-inheritance, though this is not viewed as a major drawback since multi-inheritance is not recommended to be used unless it is absolutely necessary. However, classes in Java are able to implement interfaces while extending a class. So inheritance from a non-concurrent class does not mean concurrency cannot be implemented. It just has to be done thought interfaces or inner classes. Inheritance is very useful tool when designing repetitive code; just encapsulate it into a generic superclass and extend that in subclasses. This way classes may also access into same variables, although in case of concurrent behaviour thread-safety needs to be ensured, and methods.

As activity class in Android is one of the most important classes and with a superclass extending the `Activity` class, it is a viable class for inheritance and can benefit a great deal from it. One very useful way of forcing certain behaviour to subclasses without interfaces is to declare the superclass abstract and implement there only abstract method stubs. This way all the subclasses have to implement these methods, for example a handler in superclass can enforce the subclasses to implement a callback method where the received messages can be rerouted and filtered accordingly. A good example of a scenario, depicted in figure 11, incorporating reusable code, yet offering the accuracy to route messages to a specific activity where each of the callback method filters messages according to an identification number found in `Message.what`-attribute (figure 17).

```
@Override
protected void onServiceCallback(Message msg) {
    if(msg.obj != null && msg.what == ACTIVITY_ID){
        adapter = new LogAdapter(this, R.layout.log_listitem, parseLog((ArrayList<String>) msg.obj));
        listview.setAdapter(adapter);
        progress.dismiss();
    }
}
```

FIGURE 17. Callback method.

Concurrent classes can also inherit superclasses, just as any other class, and extend it to accommodate more specific purpose while the superclass only offers a generic base to build on. A good example is a set of background workerthreads, which inherit from a single superclass extending the actual `Thread`-class and holding everything in

common with these threads and extending it to suit each one's specific needs, such as database interaction or networking.

6.3.2 Inner classes

If extending a superclass is not possible and the behaviour will stay inside a single activity, nesting an inner class is a useful way to go. Even interfaces, such as Runnable and Callable, can be instantiated from inner classes. Inner classes cannot be used from anywhere else than the outer class wrapping it. This and Android activity's lifecycle are the factors to take into consideration when using inner classes.

The inner class is able to access the outer class's methods and variables, even those declared private. (Sierra & Bates, 2005, 376) This combined with concurrent classes is risking the thread-safety and data consistency. This in mind, the variables required by the inner class should be passed to it through the constructor, just as with non-nested classes, to ensure that all the variables are processed in the same thread as they are created. Inner classes are very handy for short and local tasks, for example `Asynctask` can often be seen as an inner class.

7 SECURITY CAMERA APPLICATION

As for the constructive part of this thesis, Haltu Ltd requested a security camera application for Android to support the already existing web based application. The application requirements included a user login, camera listings and specific data for each camera, which included five the most recent images from the selected camera, log data and clearance data for that given security perimeter and naturally an update possibility for the data being viewed. As for the entities in need for concurrency, clearly networking can be identified since the user needs to log in and download data.

A tab widget was required to divide the data sets into their own dedicated activities, accessed easily by different tabs. The activities resemble each other considerably as the purpose of them all is to show data downloaded from a web service. This is why an inheritance structure seemed appropriate to avoid duplicate code (figure 18). Another reason for inheritance was to maintain thread status while switching between activities in the tabs. As the superclass activity contains all these activities and as the threads retrieving data are located also in the superclass, switching between subactivities does not interrupt the thread's progress. In other words, if the gallery activity's thread has downloaded two images out of five when the user switches to inspect the log data, the gallery thread still continues to make progress in the background and when the user comes back to gallery tab, all of the images may have been downloaded, depending on the network status.

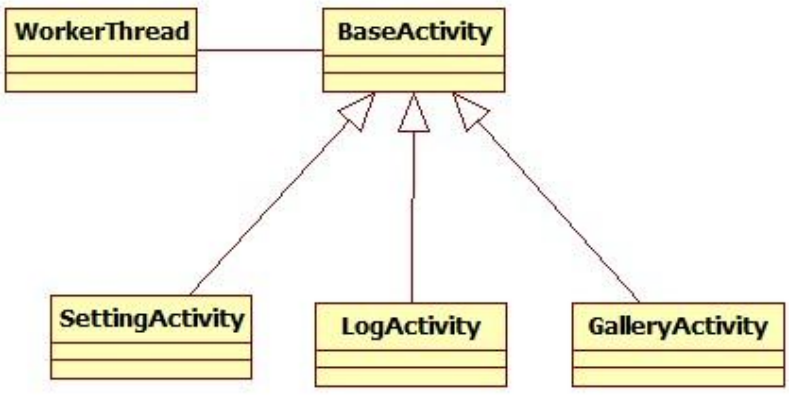

FIGURE 18. Activity structure in the security camera application
of activities accessing the WorkerThread.

The thread start is encapsulated in the abstract `BaseActivity`-class, which also contains the `Handler` from figure 12. `WorkerThread` sends its results in a `Message` which is identified by a constant variable from each activity. The `handler` receives those messages and forwards them into the callback method, which is a protected and abstract method. This enforces the subclasses to implement it. Each activity identifies messages designated for it by the constant variable mentioned earlier. The implementation of this can be viewed from figure 19.

```
@Override
protected void onServiceCallback(Message msg) {
    if(msg.obj != null && msg.what == ACTIVITY_ID){
        adapter = new LogAdapter(this, R.layout.log_listitem, parseLog((ArrayList<String>) msg.obj));
        listview.setAdapter(adapter);
        progress.dismiss();
    }
}
```

FIGURE 19. Callback method in subactivities.

Each data set from the thread, images excluded, would be standard JSON; therefore the data processing can be completed by a single generic processing method and the return values dispatched to the UI thread for publishing. Images are only converted into bitmaps and saved into a temporary cache which is removed after the `GalleryActivity` is destroyed.

Benefits of this implementation is a minimal amount of duplicate code as well as the identification of the message as the calling activity passes the variable for the thread each it call it. With this scenario, the inheritance combined with the basic implementation of thread class is working really well, especially since the `BaseActivity`-superclass is abstract, forcing the subclasses to implement specific behaviour.

As an improvement, concurrency could be achieved by using the `Runnable`-interface instead of subclassing `Thread`. Secondly the image processing could be optimized by encapsulating the input stream processing into its own thread. This would enable parallel handling of images whereas now images are handled one by one. The amount of data moved would naturally remain the same, but the time used to handle it would be decreased. Also the `Messages` used the transfer data to the UI thread could be created more efficiently, using `Handler.obtainMessage()` or `Message.obtain()` –

methods. This way the object itself would be pulled from a pool of recycled objects. This would reduce the use of memory. As the use of `Messages` in such a compact application is minimal, the use of `Message's` constructor is not causing any real issues. It would be just a good programming principle to use the recycle pool instead. (Message, 2012)

Figure 20 shows the basic graphical interface of the application. Images are pixelated due to non-disclosure agreement but the idea behind the graphical user interface remains clear. The left view with the images is the actual security camera view. Upper image shows the chosen image which is clickable and will open to a viewing activity as large as possible. Underneath it is a `Gallery` component, showing all the images with a horizontal scroll. The right view shows the graphical user interface while the concurrent download of data is executed in the background. The dialog is shown to inform the user that the chosen function is being executed and the application is running normally.

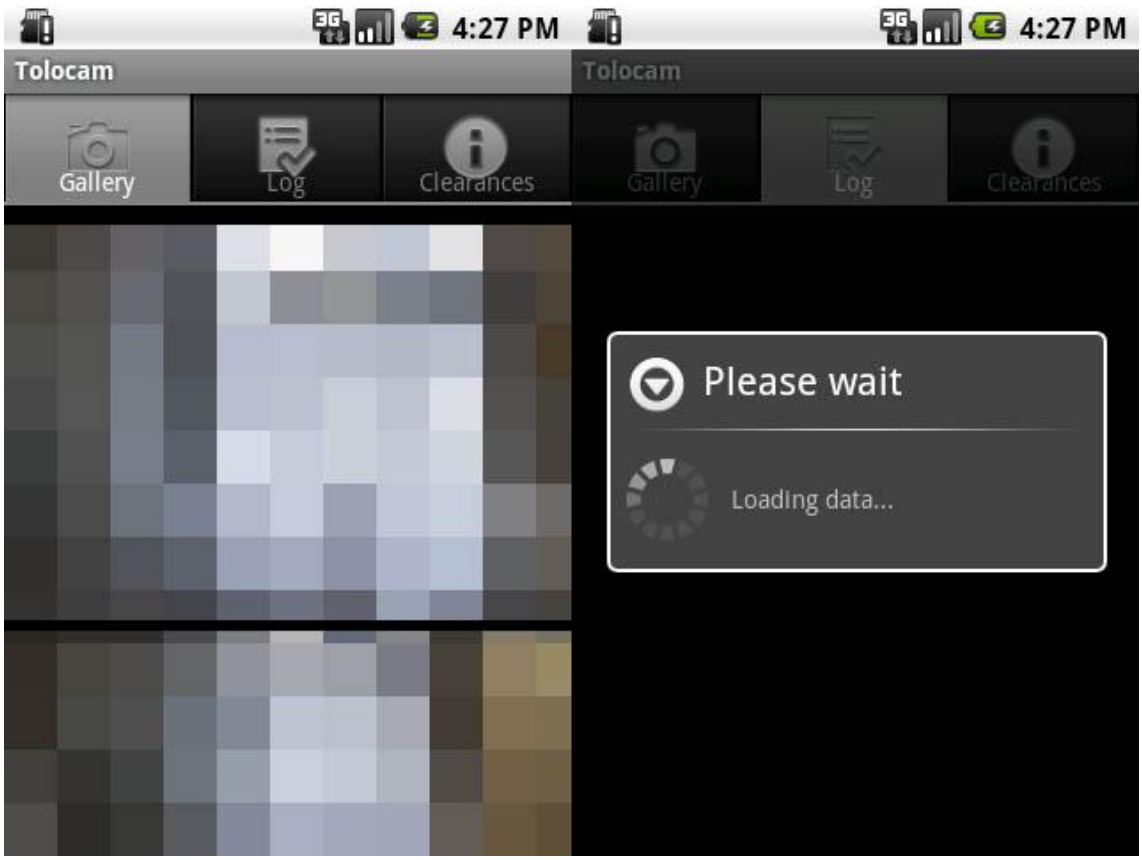

FIGURE 20. Screenshots of the security camera application's tabviews.

8 CONCLUSIONS

It has been established that concurrency is a crucial part of any software development and it can cause gray hairs if implemented improperly or a need for refactoring emerges. This is why information on different approaches and tools is crucial and may help to avoid several issues.

The main objective of this thesis was to analyze and provide an insight to the tools available for concurrency on Android and present possible issues and means to avoid them. These aspects will help to improve the whole application development process as the information found in this thesis should provide a good base for inexperienced Android developers about the guidelines and possibilities of concurrency in Android development. More experienced developers are able to reflect their development methods based on this thesis and possibly cultivate their programming routines into something more effective. However, the information has to be always applied to each specific situation individually in order to achieve the desired outcome, as any solid scenarios or definite solutions are impossible to give because they are always case-sensitive and are certain to vary. In this light, the thesis is able to achieve its goal.

The purpose of this thesis was to enhance and shape the application's design process and by achieving the set objectives, the purpose is also fulfilled, as the design process' workflow model is supporting the given insight of the concurrency in Android. This way the developers are able to proceed stage by stage on the journey of building concurrent behaviour from the identification of the need for concurrency to the actual implementation.

This thesis provided a good and up-to-date basis to learn more about concurrency, especially since multicore processors are starting to flood the markets. The future mobile devices will push the requirements of mobile applications even further, yet offering more tools and possibilities to accommodate the needs. The Android development is also evolving into something even more challenging with each new version the operating system, especially since the Android 4 is combining both, tablets and smart phones, under the same operating system. This emphasizes the need for a

good and versatile structure, which can only be achieved through a good and clear development process.

BIBLIOGRAPHY

Apple, Inc. 2011. Concurrency and Application Design. Updated 19.1.2011. Read 23.12.2011.
http://developer.apple.com/library/ios/#documentation/General/Conceptual/ConcurrencyProgrammingGuide/ConcurrencyandApplicationDesign/ConcurrencyandApplicationDesign.html

Breshears, C. 2009. The Art of Concurrency. A Thread monkey's Guide to Writing Parallel Applications. 1st edition. USA: O'Reilly Media, Inc.

Burnette, E. 2010. Hello, Android. Introduction Google's Mobile Development Platform. 3rd edition. USA: Pragmatic Programmers, LLC.

Cesarini, F. & Thompson, S. 2009. Erlang Programming. 1st edition. USA: O'Reilly Media, Inc.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. 2008. Java Concurrency in practice. 6th edition. USA: Pearson Education, Inc.

Google, Inc. 2011. Processes and Threads. Updated 4.11.2011. Read 6.11.2011.
http://developer.android.com/guide/topics/fundamentals/processes-and-threads.html

Google, Inc. 2011. Activities. Read 5.12.2011.
http://developer.android.com/guide/topics/fundamentals/activities.html

Google, Inc. 2011. TimerTask. Read 11.12.2011.
http://developer.android.com/reference/java/util/TimerTask.html

Google, Inc. 2011. Painless Threading. Read 22.12.2011.
http://developer.android.com/resources/articles/painless-threading.html

Google, Inc. 2011. BroadcastReceiver. Read 27.12.2011.
http://developer.android.com/reference/android/content/BroadcastReceiver.html

Google, Inc. 2011 Services. Read 27.12.2011.
http://developer.android.com/guide/topics/fundamentals/services.html

Google, Inc. 2012. Handler. Read 5.3.2012.
http://developer.android.com/reference/android/os/Handler.html

Google, Inc. 2012. Message Read 7.3.2012.
http://developer.android.com/reference/android/os/Message.html

Harold, E., 2004, Java Network Programming. 3rd edition. USA: O'Reilly Media, Inc.

Magee, J. & Kramer, J. 2005. Concurrency, State Models & Java Programs. 5th Edition. USA: O'Reilly Media, Inc.

Mednieks, Z., Dornin, L., Blake Meike, G. & Nakamura, M. 2011. Programming Android. Java Programming for the New Generation of Mobile Devices. 1st edition. USA: O'Reilly Media, Inc.

Oracle. 2011. Class Thread. Read 11.11.2011.
http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Oracle, 2012. Guarded Blocks. Read 8.2.2012.
http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Oracle, 2012. Processes and Threads. Read 8.2.2012.
http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

Oracle. 2011. Interface Runnable. Read 6.12.2011
http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html

Oracle. 2011. Java Thread Primitive Deprecation. Read 11.12.2011.
http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

Sierra, K & Bates, B. 2005. Head First Java. 2nd edition. USA: O'Reilly Media, Inc.

Steele, J. & To, N. 2010. The Android Developer's Cookbook. Building Applications with the Android SDK. 1st edition. USA: Pearson Education, Inc.

9 APPENDICES

APPENDIX 1: 1 (3)

```
package fi.haltu.networking;

import java.io.IOException;

public class WorkerThread extends Thread {

    private String user;
    private String pw;
    private int msg_what;
    private Context context;
    private Handler handler;
    private DefaultHttpClient http;
    private CredentialsProvider creds;
    private int threadOperation;
    private String camUrlId;

    private static int instance_count = 0;
    private static final int MAX_COUNT = 3;

    private WorkerThread(Context c, Handler h, int messageWhat, int operation, String camUrl_id){
        SharedPreferences prefs =c.getSharedPreferences(Consts.SHARED_PREFERENCES, 0);
        context = c;
        user = prefs.getString(Consts.SHARED_PREFERENCES_USERNAME, "");
        pw = prefs.getString(Consts.SHARED_PREFERENCES_PASSWORD, "");
        handler = h;
        msg_what = messageWhat;
        threadOperation = operation;
        camUrlId = camUrl_id;

        creds = new BasicCredentialsProvider();
        creds.setCredentials(new AuthScope(
                AuthScope.ANY_HOST, AuthScope.ANY_PORT), new UsernamePasswordCredentials(user, pw)
        );

        http = new DefaultHttpClient();
    }

    private WorkerThread(Context c, Handler h, int messageWhat, int operation, String username, String password){
        context = c;
        user = username;
        pw = password;
        handler = h;
        msg_what = messageWhat;
        threadOperation = operation;

        creds = new BasicCredentialsProvider();
        creds.setCredentials(new AuthScope(
                AuthScope.ANY_HOST, AuthScope.ANY_PORT), new UsernamePasswordCredentials(user, pw)
        );

        http = new DefaultHttpClient();
    }

    public static synchronized WorkerThread getInstance(
            Context c,
            Handler handler,
            int messageWhat,
            int operation,
            String camUrlId
    ){
        if(instance_count <= MAX_COUNT){
            instance_count++;
            return new WorkerThread(c, handler, messageWhat,operation, camUrlId);
        }
        return null;
    }
```

FIGURE 11. Thread class from the security camera application.

(To be continued)

APPENDIX 1: 2 (3)

```
public static synchronized WorkerThread getInstanceCredentials(
        Context c,
        Handler handler,
        int messageWhat,
        int operation,
        String user,
        String password
){
    if(instance_count <= MAX_COUNT){
        instance_count++;
        return new WorkerThread(c, handler, messageWhat, operation, user, password);
    }
    return null;
}

@Override
public void run(){
    query(threadOperation);
    instance_count--;
}

public void query(int operation) {
    try {
        http.setCredentialsProvider(creds);

        HttpGet get = new HttpGet();
        URI uri;
        Message msg = new Message();
        msg.what = msg_what;

        switch(operation){
        case Consts.CAMERA_LIST_OPERATION:
            uri = new URI(Consts.CAMERA_LIST_URL);
            break;
        case Consts.LOAD_IMAGES:
            uri = new URI(Consts.CAMERA_IMAGES_URL + camUrlId);
            break;
        case Consts.CAMERA_LOG_OPERATION:
            uri = new URI(Consts.CAMERA_LOG_URL + camUrlId);
            break;
        case Consts.CAMERA_CLEARANCE_OPERATION:
            uri = new URI(Consts.CAMERA_CLEARANCE_URL + camUrlId);
            break;
        default:
            uri = null;
            break;
        }
        get.setURI(uri);

        int timeout = 5000;
        HttpParams params = new BasicHttpParams();
        HttpConnectionParams.setConnectionTimeout(params, timeout);
        http.setParams(params);
        handleResponse(operation, http.execute(get), msg);
    } catch (URISyntaxException e) {
        handler.sendEmptyMessage(Consts.NETWORK_ERROR);
        e.printStackTrace();
    } catch (ClientProtocolException e) {
        handler.sendEmptyMessage(Consts.NETWORK_ERROR);
        e.printStackTrace();
    } catch (IOException e) {
        handler.sendEmptyMessage(Consts.NETWORK_ERROR);
        e.printStackTrace();
    }
}
```

APPENDIX 1: 3 (3)

```
    private void handleResponse(int operation, HttpResponse response, Message msg){
        try {
            InputStream stream = response.getEntity().getContent();
            switch(operation){
            case Consts.CAMERA_LIST_OPERATION:
                if(response.getStatusLine().getStatusCode() != 401){
                    SharedPreferences prefs = context.getSharedPreferences(Consts.SHARED_PREFERENCES, 0);
                    prefs.edit().putString(
                            Consts.SHARED_PREFERENCES_CAMERA_LIST, Utils.convertStreamToString(stream)
                    ).commit();
                    msg.obj = Utils.convertStreamToString(stream);
                }else{
                    msg.obj = null;
                }
                break;
            case Consts.LOAD_IMAGES:
                ArrayList<String> image_ids = Utils.parseImageIdsFromJson(Utils.convertStreamToString(stream)) ;
                int i = msg.what;
                getFootage(image_ids, i);
                msg.obj = null;
                break;
            case Consts.CAMERA_LOG_OPERATION:
                msg.obj = Utils.convertStreamToArrayList(stream);
                break;
            case Consts.CAMERA_CLEARANCE_OPERATION:
                msg.obj = Utils.convertStreamToArrayList(stream);
                break;
            default:
                break;
            }
            handler.sendMessage(msg);
        } catch (IllegalStateException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void getFootage(ArrayList<String> urls, int m) {
        try {
            http.setCredentialsProvider(creds);

            HttpGet get = new HttpGet();

            URI uri;
            for(String str: urls){
                Message msg = new Message();
                msg.what = m;

                uri = new URI(Consts.IMAGE_DIR_URL + str + "/");
                get.setURI(uri);
                HttpResponse response = http.execute(get);

                InputStream stream = response.getEntity().getContent();
                Bitmap b = BitmapFactory.decodeStream(stream);
                CacheHandler.saveBitmapToCache(context, str, b);
                msg.obj = b;
                msg.arg1 =Integer.parseInt(str);
                handler.sendMessage(msg);
            }

        } catch (URISyntaxException e) {
            handler.sendEmptyMessage(Consts.NETWORK_ERROR);
            e.printStackTrace();
        } catch (ClientProtocolException e) {
            handler.sendEmptyMessage(Consts.NETWORK_ERROR);
            e.printStackTrace();
        } catch (IOException e) {
            handler.sendEmptyMessage(Consts.NETWORK_ERROR);
            e.printStackTrace();
        }
    }
}
```