Daniel Markart

Implementing reliable Android applications

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Fernitz, October 2014

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Android applications are distinguishable from their traditional Java counterparts by many unique features. The often unpredictable life cycle due to limited resources of many mobile devices and the dependence on Web Services are only two examples.

This work will answer the question how to make a standard Android application more reliable. The big part of a test strategy in the development phase is not considered in this work though. The key subjects are runtime fault detection and event handling.

The theoretical part is split into three sections. The first one is a brief introduction into the Android architecture, the life cycle of applications and the most important components. The second section is about runtime, and service monitoring. The last section is about fault life cycle techniques, redundancy and fault injection. There is also an introduction of some Android libraries which can help to introduce reliability methodologies into the application.

The first section of the practical part is the implementation of an Android (service) monitoring library. It is explained, which Android key components are used, how they are implemented, and why they were selected in favor to their alternatives.

The second section of the practical part, is the overview of an implementation of a client server system. The client is a fully operational Android application, that uses various instances of the service monitoring library. The monitors will continuously scan the system and services for breakdowns/faults and activate event handlers when faults are detected. The

monitoring points are carefully selected specific weak points in an Android environment. The server part is a partly mocked jetty server with a Swing GUI.

Zusammenfassung

Obwohl Android Projekte in der Programmiersprache Java geschrieben sind, haben sie doch sehr viele Eigenheiten, welche sie von ihren traditionellen Java Gegenstücken abheben. Zu den Besonderheiten zählen der einzigartige life cycle der Android Programme, sowie die sehr große Abhängigkeit von Web Services.

Diese Arbeit befasst sich mit der Frage, wie man zuverlässige Applikationen für das Android System erstellt. Es geht vor allem um das Erkennen und Eindämmen von Fehlern während der Laufzeit. Auf den großen Bereich des Testens von Android Projekten in der Entwicklungsphase wurde dagegen bewusst nicht eingegangen.

Der erste Abschnitt des theoretischen Teils, ist eine kurze Beschreibung des Android Betriebssystemes. Danach wird auf runtime monitoring und service monitoring im Allgemeinen eingegangen. Der dritte Abschnitt befasst sich mit anderen Methodiken die für die implementation einer zuverlässigen Android Applikation grundlegend sind. Darunter die Theorie hinter fault lifecycle techniques, redundancy und fault injection. Des weiteren werden einige libraries aufgelistet, die den Programmierern helfen, ein zuverlässiges Programm zu erstellen.

Der praktische Teil gliedert sich ebenfalls in zwei Abschnitte. Der Erste ist die Implementierung einer (Service) monitoring library. Hier wird darauf eingegangen wie man eine library erstellt und welche speziellen Android Komponenten eingesetzt werden.

Der zweite Achnitt beschreibt den Aufbau einer Android Applikation, mit welcher veranschaulicht wird, wie die service monitoring library verwendet

wird. Es werden verschiedene Monitore implementiert, welche lokale, als auch Web Services kontinuierlich überprüfen und im Fall eines Problemes verschiedene Lösungmöglichkeiten bieten. Bei den Problemen, sowie den Lösungsmöglichkeiten, wird speziell auf Android spezifische Varianten eingegangen. Des Weiteren wurde ein kleiner Jetty Server implementiert, der den Kommunikationsgegenpart der Applikation darstellt und eine Swing GUI besitzt.

Contents

List of Figures

1 Android

Android[1] is a Linux based operating system, which was originally designed for mobile devices such as smart phones or tablets. It is developed and maintained as an open source project by the *Open Handset Alliance*[2], led by Google. At the beginning, only the company Android Inc, which was founded in 2003, was working on the project. This company was, however, already financially backed by Google. In 2005, Google bought Android Incorporated. In 2007, Google founded the Open Handset Alliance. The Android source code is released under the Apache License[3] and accessible for everyone on GoogleSource[4] or GitHub[5] [Gandhewar and Sheikh, 2010].

1.1 Android architecture

As seen in Figure 1.1, the Android software stack consists of 5 layers: applications, application framework, libraries, Android runtime and the Linux kernel. The applications layer offers frequently used and essential applications out of the box. The application framework provides important application programming interfaces (APIs) for the applications layer. The Library layer contains, a special for embedded Linux based mobile devices, tuned standard C system library, as well as media, database and security

[1] `http://www.android.com` (visited on 12/08/2013)
[2] `http://www.openhandsetalliance.com/` (visited on 12/08/2013)
[3] `https://www.apache.org/licenses/LICENSE-2.0` (visited on 12/08/2013)
[4] `https://android.googlesource.com/` (visited on 04/27/2014)
[5] `https://github.com/android` (visited on 04/27/2014)

libraries. It also contains the Android runtime [Maia, Nogueira, and Pinho, 2010 and Arzt et al., n.d.].

The bottom layer is the Linux kernel. This is a hardware abstraction layer, which provides device driver and enables the other layers to interact with the hardware. The Android version as of December 2013 (Android 4.4 KitKat) used a modified Linux kernel 3.x. Until version 4.0 *Ice Cream Sandwich*, the Linux kernel with the version number 2.6 was used.

Android uses its own Java virtual machine named Dalvik and not the original one. Dalvik is a virtual machine especially optimized for the needs of mobile devices with constrained memory and central processing unit (CPU) capabilities[6]. Dalvik just-in-time (JIT) compiles and runs *Dalvik executable* (.dex) files, which are wrapped in the Android package (.apk) file. Dalvik is an infinite register-based machine unlike the stack based original Java virtual machines. Therefore it requires less computation time, as well as 30% less instructions to finish the same tasks as the original stack based VM. Every Android application runs in its own instance of the VM and therefore has its own process. No applications have direct access to one another [Nimodia and Deshmukh, 2012 and Kumar Maji et al., 2010].

Android applications are written in Java. When the project is compiled, it is saved as an Android package (.apk) file (see Figure 1.2). Such an .apk file holds .dex files which are Java .class files converted to Dalvik byte code, as well as the *AndroidManifest.xml*, and compiled/uncompiled resources for the application[7].

Every permission that is needed for the application to work properly has to be declared in the *AndroidManifest* file. If the application tries to access an API without declaring the corresponding permission, an exception is thrown.

[6] `https://www.tbray.org/ongoing/When/201x/2010/11/14/What-Android-Is` (visited on 12/08/2013)

[7] `https://developer.android.com/tools/building/index.html` (visited on 12/08/2013)

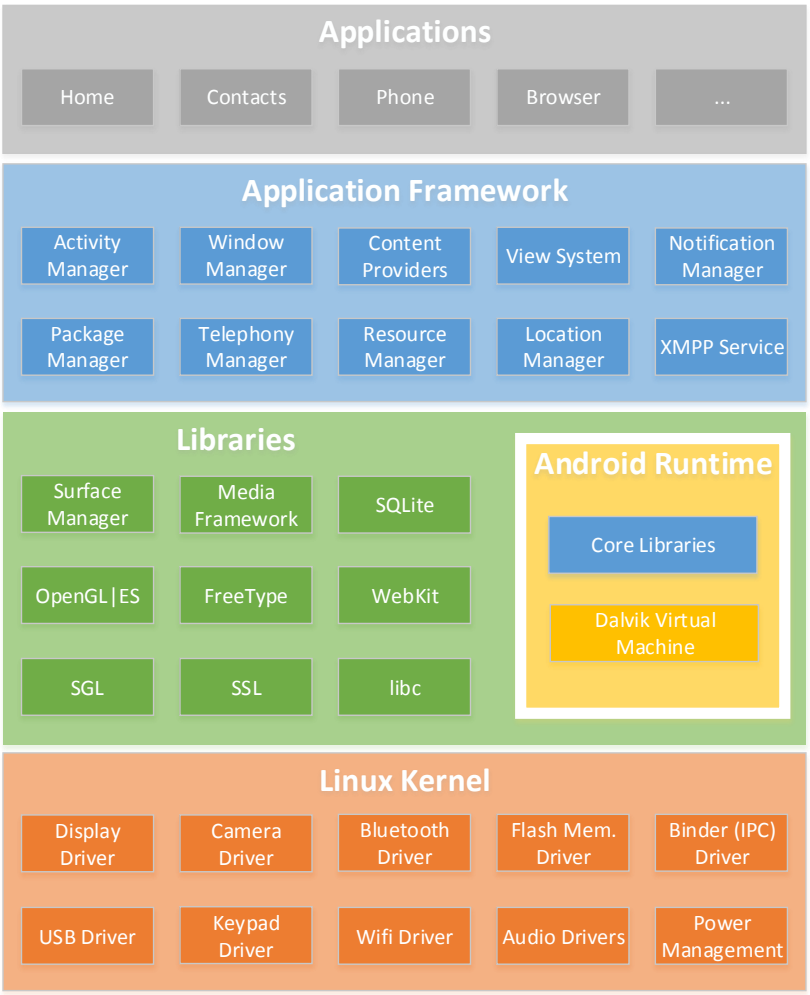

Figure 1.1: Android architecture.

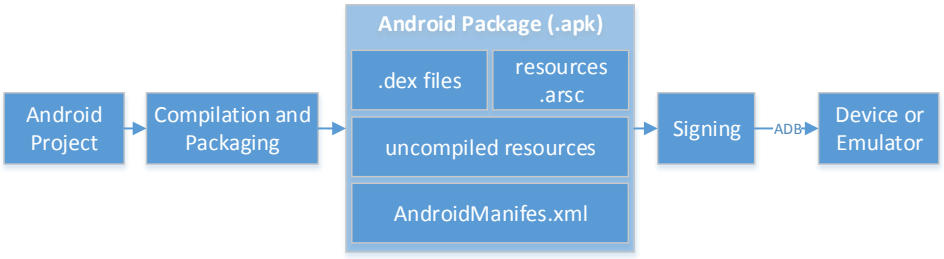

Figure 1.2: Android build process.

Android's main aim is to support the instruction set architecture *ARM*[8], as most mobile devices, such as smartphones and tablets, are *ARM* based. Support for the x86 architecture[9] is provided by the Android x86 project[10]. For example Google-TV[11] uses a modified x86 version of Android. It is also possible to run the emulator native on a x86 based computer.

1.1.1 ART

Android Runtime (ART)[12] is a new runtime, expected to replace Dalvik. It is already included as an experimental preview in Android 4.4 KitKat but not turned on by default. The Android L preview, which was released at the Google IO 2014 indicates that ART will replace Dalvik as the default runtime in this version. Instead of JIT compiling every time an application is

[8] `http://www.arm.com/products/processors/instruction-set-architectures/index.php` (visited on 12/08/2013)

[9] `https://en.wikipedia.org/wiki/X86` (visited on 12/08/2013)

[10] `http://www.computerworld.com/s/article/9222323/Google_s_Android_4.0_ported_to_x86_processors` (visited on 12/08/2013)

[11] `http://www.google.com/tv/` (visited on 12/08/2013)

[12] `https://source.android.com/devices/tech/dalvik/art.html` (visited on 12/08/2013)

started, like in Dalvik, ART uses ahead-of-time (AOT) compilation when the application is first installed. This turns them into truly native applications, which can be executed without JIT compiling. In theory this process will speed up many application and reduce the overall power consumption. One drawback is that applications need more space on the internal storage [Hall and Anderson, 2009].

1.2 Application life cycle

Many Android devices have limited memory capabilities. Therefore, the Android system manages low memory by terminating unneeded running applications without consultation of the user. This means that the system is allowed to kill a running background process at any time.

For every running application, Android creates an Application object. This is then started in a new process, with a unique ID under a unique user. The programmer is able to extend this class and declare it in the AndroidManifest, otherwise Android creates the default object [Burnette, 2009].

The Application object is alive as long as any other component of the application, like an Activity or a Service, runs. Such an object has the following life cycle methods:

- onCreate() is the first method of the application to start before all other components
- onLowMemory() is called, when the Android system decides that this application should be terminated due to low memory
- onTerminate() isn't called in production and only used for testing
- onConfigurationChanged() is called whenever the configuration changes (i.e. when switching from portrait to landscape mode)

If Android runs low on memory and needs to terminate some processes, it does that with respect to the following priority system.

Process status	Description	Priority
Foreground	A foreground application, is an application where an activity is active, and the user is interacting with this activity. Also if an onReceive() function of a broadcast receiver is running, or when a service runs one of his life cycle methods.	1
Visible	This is, when an activity is active but the user isn't interacting with it.	2
Service	The application has a running service, but no current activity.	3
Background	Applications are in the background, when they are still in a least recently used (LRU) list of the Android system, although they have no current activity or a running service.	4
Empty	This is an application without any active components.	5

Table 1.1: Android life cycle priorities

1.3 Market share and usage

As of September 2014 it is claimed that the market share of Android for smartphones has reached 83%[13]. According to Google, more than 1 billion devices were already activated running the Android operating system[14].

In addition, Android is not only installed on smartphones and tablets. Due to its open source approach, it is easier to port to new hardware. There are many more products, which use Android as their operating system. *Google Glass*[15] is running a slightly modified version of Android 4.0. There are also notebooks, netbooks, smartbooks, ebook readers, wristwatches, headphones[16], Android operated game consoles, cd and dvd players, satnav systems, home automation systems, mirrors, cameras, landlines, treadmills and even refrigerators that run Android as their main operating system.

1.4 Android terminology

The Android operating system defines some unique elements, which are utilized by nearly all applications [La and Kim, 2009].

Context An Android context is an interface to global information about an application environment, which is needed to load resources, launch a

[13] `https://www.idc.com/prodserv/smartphone-os-market-share.jsp` (visited on 09/28/2014)

[14] Rossignol, 2013.

[15] `https://www.google.com/glass/start` (visited on 12/08/2013)

[16] `http://www.theregister.co.uk/2011/01/12/now_audio_admiral_touch/` (visited on 12/08/2013)

new activity, get a file path, create a view or obtain a system service.

Activity Every time the application is visible on the display of the device an activity needs to be active. The activity shows the user interface (UI), which enables a person to directly interact with the application. Most of the time activities show the application in a full screen window to the user. Each activity and the corresponding base context is unique in the application instance.

Fragment The fragment is a part of the UI and can only be shown in an activity. Often the activity holds only one fragment. If that is the case, the fragment is filling the whole screen. They were originally introduced to help scale the applications to larger screen sizes, as the activity can show more than one fragment at a time (e.g. the smartphone version of the application shows one fragment and the tablet version two).

Service A service has access to the resources of the application, like the activity or a fragment, but is used to perform a longer-running operation without interaction with the user, much like a Linux Daemon. It also doesn't supply functionality for other applications to use. As long as a service is running, the application process is alive. To be able to use a service, it must be registered in the Android manifest.

BroadcastReceiver This is a base class, which enables the application to receive intents sent by *sendBroadcast*. It can also catch system based broadcasts, for example when the device is started or shut-down or when the device gained access to the internet. A *BroadcastReceiver* can be registered static or dynamic. Static receivers can receive an intent even if the application isn't running, but dynamic receiver will be unregistered when the application is closed. Broadcasts sent with *sendBroadcast* are system wide but can be

equipped with permissions. It is also possible to send local (application level) broadcasts with the *LocalBroadcastManager*[17].

Intent Intents are messages used to activate services, activities and broadcast receivers. They can be seen as an 'intention', which holds bundles of information to perform an action. In the case of broadcast receivers1.4, they are holding information about something that happened.

ContentProvider A content provider provides access to a structured set of data. They are the standard connectors between data of two processes.

AndroidManifest An *AndroidManifest.xml* file is an essential part of every Android application.

- It provides core information about the application.
- It states the Java package.
- It describes the application, the activities, the services, broadcast receivers and content providers.
- It declares permissions, which are necessary to use the application and which permissions are necessary for others to interact with this application.
- It declares the minimum Android API version. Devices with a lower level cannot run this application.

Permissions They are one of the most important security features of Android. Android is a privilege-separated operating system. This means that every application can only use APIs, if the corresponding permissions are set in the *AndroidManifest* file. Every application has a distinct system identity. Therefore, every application is isolated from other applications and the rest of the system.

[17] `https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html` (visited on 01/04/2014)

AlarmManager The Android *AlarmManager* has access to the alarm services. With its help, the application developer is able to schedule the execution of code in the future. An intent is used to register a new *AlarmManager* and when the alarm goes off, this intent is broadcasted by the system. If the application registers a static receiver for this intent, it is possible for the *AlarmManager* to start the application, if not already running, and execute the code placed in the receiver.

GCM Google Cloud Messaging (GCM) for Android allows a server to send data to any Android device. The server doesn't need to hold a port open, or know the IP address of the device, but the device has to register it's own GCM ID at the server. After sending a message via GCM, the Google cloud services automatically know how to find the device with the corresponding GCM ID.

2 Runtime Software Monitoring

Runtime software monitoring is an approach to analyze the state of a running system. It is generally used to add a defense layer against catastrophic failures in programs and to support state information in tests. But it is also used in other cases such as

- security or safety policy monitoring
- debugging
- testing
- verification
- validation
- profiling
- fault protection
- behavior modification (e.g., recovery)
- performance analysis
- software optimization
- diagnosis

Sometimes, a monitor provides information about the fault to the user and guides the system to recover from faults or even revert the faulty states.

The execution of a monitor can be either time-driven or event-driven [Albari, 2005]. A time-driven monitor (called sampling) collects information about the execution synchronous or asynchronous to the normal execution state. If asynchronous, information is requested by the monitoring system itself. Such an approach is insufficient for a behavioral analysis, as it only provides a summary of statistical information of the execution process. An event-driven monitor (tracing) collects all occurrences of an event regardless of a time interval. Thus it often creates a larger volume of data, than the

time-driven approach.

2.1 Fault Detection Monitor

A fault detection monitor can prove, if the behavior of an application complies with a pre defined property. The developer needs to specify formal requirements of the application from whom the monitors are often synthesized. A fault detection monitor only analyzes one or a few execution traces. Therefore, it is by far not as 'complete' as other approaches try to be. It also works directly with the actual system, circumventing an error prone and very difficult step of formally modeling the whole system. [Delgado, Gates, and Roach, 2004 and Wikipedia, 2013].

In contrast, testing, model checking and theorem proving are aimed to ensure universal correctness of programs. These traditional verification techniques are much more complex and very difficult, time consuming, and expensive to implement. Of course, a monitor cannot replace them in general, but adds an additional reliability layer and could replace some very difficult to implement parts of a testing system [Wikipedia, 2013].

2.1.1 Basic Monitor Layout

Figure 2.1 shows that a monitor basically consists of 2 parts, the observer and the analyzer. The observer checks the state of the executing software either by getting the state automatically when some declared events are triggered, or by pulling them. The developer has to provide the monitor with a set of properties. These properties decide which states are correct and which of them are faulty. Every state is sent from the observer to the analyzer, which compares them to the pre defined expected values. If an evaluated state does not meet the properties, the state is automatically sent to the event handler. The event handler then has the mission to cope with the faulty state. The result of the positive monitoring is mostly written into

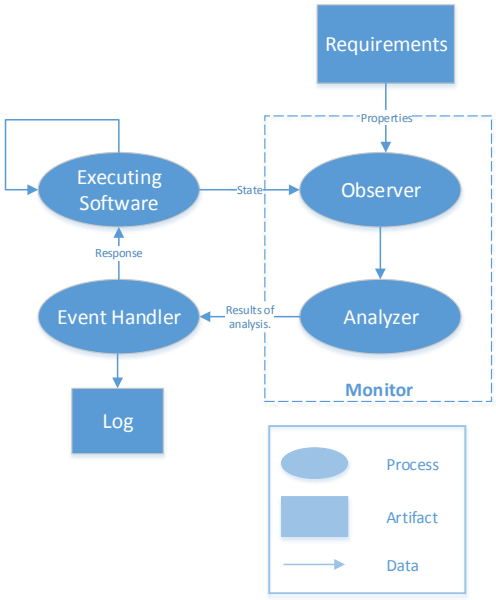

Figure 2.1: Monitoring Architecture.

a log for the developers. There are different possibilities to respond to a violation:

- Inform the user.
- Halt the system.
- Recover from or revert the faulty state.
- Memory rollback.
- Exception handling.

Event handlers have different response effects. A response effect states how the normal execution process can be altered by the monitor. Some event handlers just create a log and have no effect on the behavior of the program. Other monitors can set a break point and let the user decide how to continue, these monitors are called 'user controlled monitors'. If the event handler performs a recovery strategy by itself, it is an automated system.

It is very common in practice that the monitored system is heavily modified, to be able to use a monitor effectively. In these cases, the monitor and the monitoring system are often not as nicely separated as shown in Figure 2.1. Then they are fused to one interleaving system and the running system deals with tasks the monitor normally should handle. There are of course different possibilities to implement a monitor. The main characteristics of every monitor are, as hinted in Figure 2.1, monitoring points, placement, platform, and implementation. The next sections will provide a short introduction into these characteristics [Delgado, Gates, and Roach, 2004].

2.1.2 Monitoring Points

These are the points in the execution program, where events are sent to the monitor and the monitoring code starts its execution. The actual points can be classified manually or automatically. In the manual classification, the developer has to include the code manually at the desired places of the source code. The automatic classification is performed by tools, which detect those points by themselves.

2.1.3 Placement

The placement decides from where the monitoring code is executed. It is either inline or externally performed. Inline placement means that the monitoring code shares the resources of, and is called by the executing program. The monitoring code, including observer, analyzer and event handler are placed directly in the source of the executing program. An offline placed monitor doesn't share its resources, as it runs in a distinct thread or process. It is even possible to place this kind of monitor at a separate device. If the program doesn't have to wait for the analyzer to finish before continuing the main task, it is an offline asynchronous monitor. If it has to wait, it is an offline synchronous monitor.

2.1.4 Platform

Monitors can be either implemented in the software as source code or they can be a piece of hardware. Hardware monitors are for example a microprocessor with sensors attached to the target system.

2.1.5 Implementation

The implementation characteristic states in which processes the monitor is running.

- Single Process: The monitor runs in the same process as the target machine.
- Multiprogramming: The monitor is executed on the same processor but a different process or thread.
- Multiprocessor: The monitor and the monitored program run on different processors.

2.1.6 Example: Monitoring Oriented Programming

An example for a generic runtime fault detection monitor framework, is Monitoring-Oriented Programming (MOP)[1]. In MOP, the developer specifies properties which are automatically synthesized into runtime monitors. These monitors, as well as the dedicated user defined handling code, are merged with the target application on compile time to check the dynamic behavior on runtime [Delgado, Gates, and Roach, 2004].

For Java applications, JavaMop was created [F. Chen and Roşu, 2005]. As JavaMop calls some classes, which are not part of the Android runtime, it is

[1] `http://fsl.cs.illinois.edu/index.php/Monitoring-Oriented_Programming` (visited on 01/17/2014)

not compatible from the get-go, but [Falcone, Currea, and Jaber, 2013] and [Falcone and Currea, 2012] demonstrated that it can be modified to work on Android with little effort.

JavaMop uses the syntax of AspectJ[2]. The principle of MOP is not necessarily program verification, but to avoid the verification of an implementation against its specification by altering the flow at runtime to not let it go wrong in the first place. JavaMop supports many different logical formalisms, in which a system behavior can be described:

- ERE (extended regular expressions)
- FSM (finite state machines)
- FTLTL (future time linear temporal logic)
- CFG (context free grammars)
- PTLTL (past time linear temporal logic)
- ptCaRet (past time linear temporal logic with calls and returns)

MOP will then translate this specification into generic monitor code. JavaMop then translates it to AspectJ code. The event handling code can then be written in Java by the user.

For example, Listing 2.1 shows the specification of a SafeLock in JavaMOP. Here is described that every thread has to release a lock which was acquired. This specification consists of 5 parts. The header states the modifiers, the parameters and the boundary, in within this property is monitored (in this case any method defined in the Main class). The second part is the declaration of 2 variables: *acq_count* and *rel_count* which are important for this monitor. The third part is an AspectJ declaration of 2 events: the acquisition of a lock and the release of a lock (acq and rel).

```
1 unsynchronized decentralized SafeLock(Lock l; Thread t)
2   within Main: * {
3     int acq count; rel count;
4     event acq after(Lock l; Thread t) :
5       call(* Lock:acquire()) && target(l) && thread(t)
6       {++ acq count; }
```

[2] `https://eclipse.org/aspectj/` (visited on 01/05/2014)

```
7       event rel after(Lock l; Thread t) :
8         call(* Lock:release()) && target(l) && thread(t)
9         {++ rel count; }
10      cfg : S   > S acq S rel j epsilon
11      @violation{
12        System:out:println(acq count + \ acquires and "
13          + rel count + \ releases at line " + Loc);
14      }
15  }
```

Listing 2.1: SafeLock in Java MOP.

The fourth part states the used logical formalism. In this case it is *cfg* (context free grammars). The last part is the event handler, this block is called if a violation of the specification was detected on runtime [F. Chen, D. Jin, et al., 2009].

2.2 Service Monitoring

The scope of the practical part of this work will answer the question how to implement an Android service monitor. As most Android applications are highly dependent on online services, monitoring them gives the system a chance to cope with a defect. To continuously monitor a service and sending error reports to the persons in charge even if the application isn't using the service at the moment, enables administrators to fix the problems as soon as possible or lets the event handler of the application cope with this problem.

Service Oriented Architecture[3] is a frequently used architectural style, especially in mobile applications. In such a system, there are many dependencies between services but every service stands for its own or in the most cases are even completely uncoupled from the program or the device [Papazoglou, 2003].

[3] `https://en.wikipedia.org/wiki/Service-oriented_architecture` (visited on 01/19/2014)

As different web services often have the same functionality and interface, they are nowadays prepared to be switched automatically, if one of them fails. This is achieved with technologies like UDDI [Bellwood et al., 2002] or WSDL [Christensen et al., 2001]. For other services, this kind of automatic fail save functionality must be implemented by hand. [Ameller and Franch, 2008]

Often, online services are used to send messages to the application. Therefore, the application does not try to connect to the service by itself. In these cases a service monitor is even more important, as the user expects messages automatically to be sent to the device.

The life cycle of an Android application differs from the ones on a traditional computer (see Chapter 1.2). If the application is solely waiting for the user to interact with the user interface and nothing else (no Android service is running and keeping the application alive), the Android system will eventually kill the running process automatically if the activity isn't active. This happens after around 40 minutes on a Nexus4 with KitKat[4] and no particularly high memory usage. This is due to the fact that Android applications are often broadcast driven in contrary to applications on a normal computer, where they will just stay alive, until the user decides to shut them down.

2.2.1 What to monitor

Figure 2.2 represents a web service quality model based on ISO/IEC 9126 [Standarization, 2001]. This model foremost shows the technical characteristics of a high quality web service.

Not all of these characteristics can be monitored. Many of them can be evaluated and measured in some way, but reliability, portability, usability and maintainability can not be monitored because they are software design

[4] `http://www.android.com/kitkat/` (visited on 03/16/2014)

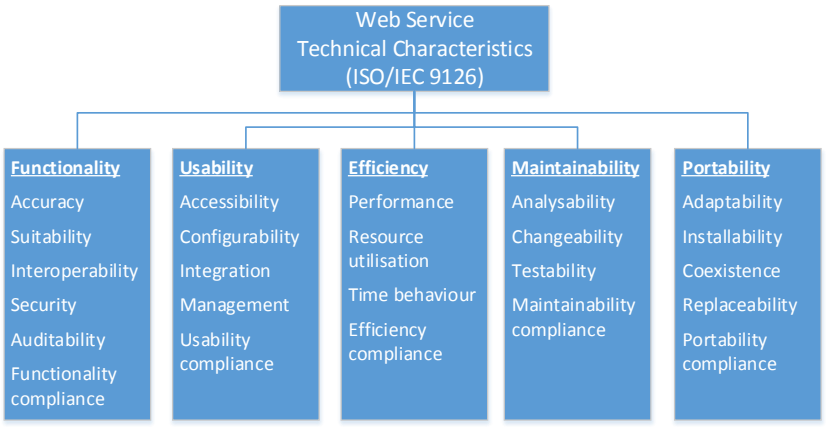

Figure 2.2: Web Service Quality Characteristics.

characteristics. Therefore, they simply won't change while the software is in a running and executing state.

Only a small subset of the remaining stated characteristics is actually monitorable. Especially accuracy, time behavior and availability are the key characteristics to be monitored in a service monitoring system.

2.2.2 Actual monitoring

There are many pieces of information necessary to reasonably monitor the accuracy of a service. Many services are complex entities, it is often not sufficient to just send one simple request to actually prove accuracy. The developer needs to know the concrete functionality of the service and often has to implement sophisticated tests to actually confirm if the delivered data is accurate.

To monitor the characteristic availability, it is sufficient to ping the service or to send a simple request. By testing and confirming accuracy, the availability check is already included, as the service must be reachable when trying to

confirm its accuracy.

The last characteristic to check is time behavior. The most important part there is the response time. For most services, it is sufficient to count the time necessary for the service to fulfill one request. In a mobile device environment, the problem with response time monitoring of server side services, is the ever changing network speed when the device is not stationary.

All server side monitoring tests depend on the connectivity state of the mobile device, which can change from one second to another. For many monitor implementations a generous connection timeout can be set.

2.2.3 Service Level Agreement

This is the part of a service contract where the service is formally defined. service level agreements (SLAs) provides information about the availability, serviceability and performance attributes of a service. Sometimes other information like billing, which are also important for customers are included in a SLA. An already existing SLA can be the basis for well defined requirements in the service monitoring environment. [L.-j. Jin, Machiraju, and Sahai, 2002]

2.2.4 Service Monitor Layout

The basic layout of a service monitor (see Figure 2.3) is very similar to the layout of a fault detection monitor described in Chapter 2.1.1. The main difference is that the executing program not necessarily commands the monitor and doesn't give its state to the monitor. Only if the analyzer part of the monitor evaluated that one service is in a malicious state, an information exchange with the target program takes place. Then the analyzer sends a message to the event handler. The event handler, which is already part of the executing program, decides how to handle the problem. The service

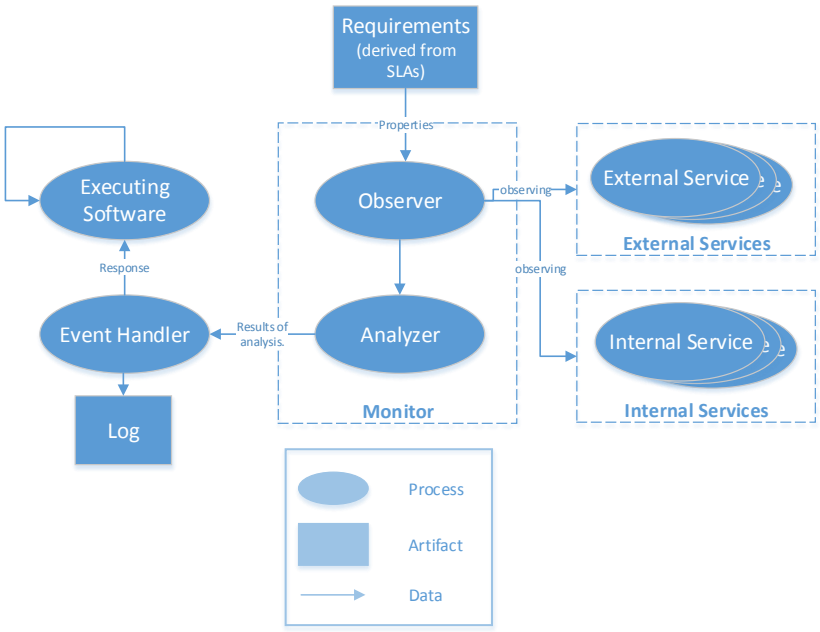

Figure 2.3: Service Monitoring Architecture.

level agreements for each service can be set via the requirements. For web services a WSLA [Keller and Ludwig, 2003] can be used.

2.2.5 Test/Monitor the Monitoring

In practice, monitoring often doesn't work as expected in a case of emergency, because it stopped unnoticed long ago. This often happens, when the monitoring itself is not monitored or tested properly. In daily use, nobody will notice that the monitoring system experienced an error, or is running in a faulty state, as the event handler normally only starts his work in the occurrence of special cases like a failing web service. Therefore, an application can work as expected for months or even years despite having a defective monitoring system and without anyone noticing.

An important technique to improve the reliability of the monitoring system is to implement automatic tests. Usually, after every change in the source code, the programmers should start a test suite, which verifies the functionality of the whole program. These tests should check if the event handlers are working as expected, if the observer behaves like expected and if the monitors are even starting. If the observer triggers the event handler, can be confirmed by simulating certain events. The observed services can be mocked or stubbed in the test suite for the test runs, these mock-ups can simulate specific events, which are not occurring frequently in the production environment.

There is a variety of different mock-up possibilities for Android and Java respectively. Many free libraries are available, which ease the use of mock-ups. Even more complex structures like an HTTP-server can be mocked with nearly no effort, when using existing libraries like *MockHttpServer*[5].

Furthermore, there are other mocking frameworks like EasyMock[6], which can make a mock object out of every available object. After the creation of such a mocking object, the developer can set the return values and mocking behavior of the object when calling the available functions.

A drawback in only running standard tests of the monitoring system is that they don't show the long term reliability of the system. What if the monitoring system stops working after the 200th instance? Nobody would notice this failure until an emergency occurs. A remedy for this problematic case would be to actually monitor the monitoring. If this is implemented properly it can be seen as an additional reliability layer. Yet, monitoring the monitoring system could proof to be difficult to implement, because it could suffer from the same problems as the actual monitoring itself. Therefore, the implementation of the backup monitoring has to be different than that of the main system. A compromise would be to implement the monitoring system in such a way that if it crashed, it can send a report to the server, but sometimes implementing this in such a way simply isn't possible. To

[5] `https://github.com/vladimirvivien/workbench/tree/master/android-tutorials/MockHttpServer` (visited on 02/28/2014)

[6] `http://easymock.org/` (visited on 02/28/2014)

send reports to the server if it works like expected (heartbeat[7]) is maybe not a bad idea to test the reliability of the monitoring system in the beta phase. It is also easy to implement but adds to the overhead a monitor is causing anyway. Also, every user would have to be registered, or it wouldn't be clear which device is the source of the heartbeat, and it would often be unclear if the monitoring stopped, because it actually failed or the user just deactivated the device, or if it went out of reception. A similar solution available is to implement a second monitor on the same device, which expects a heartbeat of the first monitor.

A possibility to verify the long term reliability of the monitoring system is to roll out a modified application just to the developers, the staff and test users. This modified version would log and maybe send detailed reports about the monitoring and the system. The biggest drawback in this solution is the limited variety of hardware and triggered use cases from the developers. A real user would certainly use the application in another way. Also, it is a very time consuming and expensive way to confirm reliability of the system.

Usual unit tests could simulate a test case for long term reliability, but it is hard to simulate all the different states a mobile device might go through. The normal usage of an Android mobile phone includes restarts, receiving calls and running many other applications concurrently. There are for sure some special events the developers will not think about when implementing a sophisticated automated test run.

The subjects monitor testing methods and reliability testing is explored in greater detail in Chapter 3.3.

[7] `http://en.wikipedia.org/w/index.php?title=Heartbeat_(computing)&oldid=582832665` (visited on 02/26/2014)

2.3 Monitor Power Consumption

State of the art (as of 01/18/2014) mobile devices have powerful processor units with up to 8 cores, high resolution displays and up to 3 gigabyte of *RAM*, which often surpasses desktop computers of the recent past. Still, they are often extremely slim and petite devices, where a big battery doesn't find a place. Therefore, the power needs of current mobile devices outgrow the moderate developments in the battery sector [Powers, 1995].

It is important that a monitor doesn't consume too much energy. Otherwise it will hurt the user experience of the application and the whole device. Then the developers are tempted to remove the parts of the application that consume the most power in order to improve the general user experience.

2.3.1 Energy Efficient Service Monitoring

In service monitoring, where the monitors are often responsible to evaluate the state of a web service, offloading some parts of the monitoring to an external server could be considered (see Figure 2.4). Instead of sending many requests to a web service to check the availability and accuracy (on the left side of the figure), a server could handle that task (right side of the figure). This approach divides the monitor into a local and a remote part. The Android application only requests the state of the web service from the server. This partition of the monitor system won't work if no data connection is available, but as it is favorably used for checking web services this doesn't matter as no web service would be reachable without a connection to the internet anyway. Sending large data volumes across the network should be avoided, as this can negate the energy saving effect of this approach [Kwon and Tilevich, 2012].

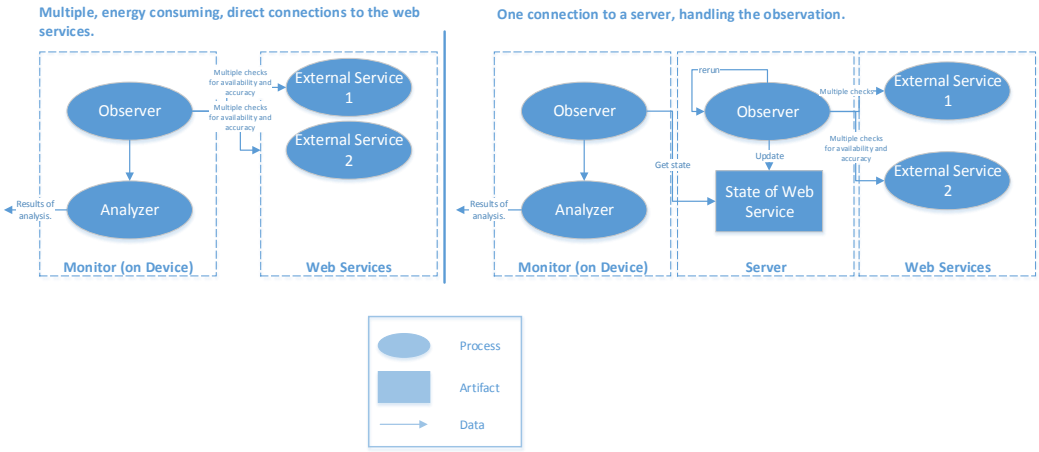

Figure 2.4: On/Off Device Service Monitoring difference.

2.3.2 Outsource Web Service monitoring completely to the Server with GCM

Another approach to improve the energy efficiency of web service monitoring, is to combine the benefits of an outsourced monitoring by the server and the capabilities of GCM. Then the Android application doesn't have any web service monitors at all. The server will monitor all the web services of interest like in Chapter 2.3.1. It will only send a message via GCM to the clients when a problem with a service is discovered. As this solution doesn't need the application to take action apart from waiting for a GCM broadcast, it is the most energy saving approach by far. The GCM part is handled by the Android device itself and active by default. Many often pre-installed Google applications like hangouts and G+ need this service to function properly. The new versions of GCM rely on the Google Play-Services.

The biggest downside of this solution is that there are no monitors for the server and the GCM service. A monitor for the server on the application would render the energy saving useless. The server could be monitored by another device which will inform the administrator, the clients however won't be informed.

If the GCM service goes down, the server must find a way to inform the application, as the client will not contact the server by itself. An easy way to inform an Android device apart from GCM is to send a SMS message. This message could be parsed by the application and it could start to contact the server by sending HTTP-requests (see Chapter 6.2.3 for further information).

2.3.3 General Monitor Methodologies for Energy Efficiency

The monitor shouldn't run in an Android service, which only exists to keep the application or the monitor alive. The Android system itself should schedule the monitor executions. For example the Android *AlarmManager* is able to take over this work by scheduling events. This service is very precise and extremely reliable, it can even wake a shut down device up and perform the specified tasks.

It is possible to deactivate web service monitoring, while no network connection can be established. The Android System sends broadcasts to all applications by default, when the network status changed. After such a broadcast, the monitoring can be continued.

If the monitoring system interchanges big data loads with the server, a possibility to save energy is to wait until the device is in an area with a fast information transfer rate, as this will also save some energy. The energy consumption depends greatly on how long the connection to the server is active. Generally, the shorter the connection to the server stands, the lower will be the energy consumption, regardless of the used technology (like 2G, 3G, 3.5G, 4G or WIFI). Android has a simple API to find out the connection type and the current speed of the connection (see Listing 2.2). [Wang and Manner, 2010]

```
1 NetworkInfo info = Connectivity.getNetworkInfo(context);
2 int connectionType = info.getType(); //WIFI or Mobile
3 int subtype = info.getSubtype(); //EDGE, GPRS, HSDPA, UMTS ...
```

Listing 2.2: Connection and connection speed in Android.

On June 24 2014 at the Google IO, Google introduced Project Volta. The main goal of this project is to boost battery life on Android devices. Volta touches many parts of the system, one of the main components is the new *JobScheduler* API. As access to the network takes a lot of energy, this API is able to bundle workload heavy tasks from all applications on the device. These bundled tasks will be executed if certain conditions are met. Furthermore, it won't attempt to establish a connection to the network if no reception is available. The downside is that the developer has no exact control about when exactly the tasks are executed. Therefore, the *JobScheduler* is predestined for application requests which are not high priority.

2.4 Common Monitoring Problems

Volume. Especially for debugging purposes, a monitor often collects massive amounts of data. This data must be processed, saved and presented. The processing filters the information and only the data of interest should be saved. The processing and saving of the data should be implemented in consideration of an efficient presentation medium. Data is useless if it is not presented in an efficient and understandable way.

Intrusion. A complex monitoring system often alters the execution environment. Then the CPU time and the communication channels are changed. The programmer must consider these alterations and implement the monitoring system in a way, which doesn't disturb the main program as the monitor shouldn't be noticeable by the user.

Access. The monitor requires access to structures and variables of the program. This could lead to inconvenient problems.

- The monitor and the program could be separate programs, then access restriction must be considered.
- The developer doesn't want to change the program to the point of merging the sources of monitor and monitored code into one inseparable blob. It is often a balancing act to keep a clear structure as well as getting access to important system variables.
- Dependent on the situation and the used architecture, it could also lead to performance issues.

[Albari, 2005]

3 Methodologies for a reliable system

3.1 Fault life cycle techniques

There are four general fault life cycle techniques, which were proposed to counter software reliability engineering problems. [Lyu, 2007]

- Fault prevention
- Fault removal
- Fault tolerance
- Fault/failure forecasting

3.1.1 Fault prevention

Fault prevention is, like the name suggests, the avoidance of faults. This is the initial step against software unreliability. This is also, in some kind or another, the goal of every software engineering methodology. General approaches of this technique are:

- Formalization of the software engineering process
- Program verification
- Early user interaction
- Requirement refinement
- Refined and systematic software reuse

- Enforced programming principles

3.1.2 Fault removal

The next step after fault prevention, is fault removal. The bugs and faults, which slipped past the fault prevention mechanism and which were already injected into the software, can be traced and fixed. The two standard approaches are software testing and software inspection.

3.1.3 Fault tolerance

When the software is released and a fault survived the prevention and the removal steps, fault tolerance is the last defense before a fault makes the transition to a program failure (see more about the transition from a fault to a failure in Chapter 3.3.2). Fault tolerance is the attribute of the system to cope with, and to survive faults. As well as to keep delivering results despite of the malicious behavior of some components. These techniques enable software programs to:

- Hinder software faults from becoming active. For example to prevent illegal operations or the confirmation of valid input and output conditions.
- Treat failed operations with a reasonable exception handling and hinder them from leaving a confined boundary.
- Activate rollback mechanisms to recover from erroneous conditions.

3.1.4 Fault forecasting

Fault forecasting is the prediction of likely faults. This makes it easier to remove them or to diminish their effects on the system. The forecasting

methods involve:

- To forecast the fault/failure relationship.
- The comprehension of the operational environment.
- Creation of software reliability measurement mechanisms.
- The evaluation of the results of the measurements.

3.2 Redundancy

Redundancy is one of the most important techniques for implementing fault tolerance, fail safety or a backup system. Redundancy is the duplication of critical systems and can be applied in many different parts of a project or infrastructure, such as hardware, software or information.

3.2.1 Information Redundancy

The event handler in Chapter 2 is a good example for this technique. This approach is also called error detection and correction methodology. In this case, the runtime monitor (the observer), detects errors and the event handler corrects them if possible, by using redundancy. An example for this approach is shown in the implementation of the monitor in Chapter 6.2.3. There, the data delivery is switched to the SMS service from HTTP if no internet connection is available.

3.2.2 Hardware Redundancy

There are different approaches for hardware redundancy. One of the most common is dynamic redundancy (stand-in spares). There, a second, mostly identical, piece of hardware, which can take over the workload if the first piece has a failure, is obtained and installed. As this work focuses on

Android software projects, there isn't much to achieve on the client side (the smartphone or other Android device), which is in the hands of the user. The server side however, can be supplied with various options:

- Server infrastructure
- User data (losing this data could upset many customers)
- Power supply (even with a second server, without power everything will come to a hold)

It is favorable to not install the backups in the same place as the main infrastructure. For example, a fire or an earthquake or any other disaster could also destroy the backup. These days, many companies outsource their server infrastructure to other companies (like Google or Amazon). Then, the responsibility of creating a redundant system lies with the company providing the infrastructure.

Another form is static redundancy (masking). There, the faults are masked by providing a majority gate, whose output represents the most likely result by comparing all inputs and selecting those which occurred most often. An example is the triple modular redundancy (TMR). In the TMR, the outputs of three logic modules flow into the majority gate. As long as only one module delivers wrong results, the output of the majority gate will be correct.

The third form is hybrid hardware redundancy. This approach combines static and dynamic redundancy. It contains a majority voting system at the core, and if the majority gate finds modules which deliver wrong results, a spare automatically takes its place. [Su and Ducasse, 1980]

3.2.3 Software Redundancy

In nearly every software project, reliability of software is attained with the help of software failure avoidance (or intolerance), which is then verified by tests. This is a well documented, comparatively good working and not too

expensive method. That said, it is still very hard to attain software failure avoidance in large or medium sized projects, nearly all of them fail to reach this goal.

Software redundancy is a different approach. If a function delivers wrong results due to programming errors, another function will take over to achieve reliability. In contrast to hardware, where physical failures predominates, software defects are time-invariant defects. This means that the same function cannot be used for redundancy because the second function would in most cases deliver the same wrong results as the first. Multi version programming for example, is a way to attain software redundancy. In this work, *Recovery Blocks* and *N-Version Programming* will be briefly introduced. Both of them are implementations of the multi version programming concept.

N-Version Programming

An approach to achieve software redundancy is *N-Version Programming*[1]. Here, separate team members or teams implement the same part of the software independently with the same initial specifications. With the goal in mind that these two parts can be used as redundant functions in the program. The teams should work independently, although, they should discuss their individual approaches to a certain degree, otherwise it is possible that they could use the same algorithms, libraries and languages to achieve their goals.

When the program is running, a supervisory program called a *driver* is needed which checks, like a monitor, the results of each part and activates the redundant functions if needed. That said, this solution is very hard to implement as it consumes much time and it is often very difficult or impossible to verify if functions in the program are delivering wrong results on runtime. Also, the initial specifications must be well defined, correct, complete, and unambiguous, or all the implementations of the redundant

[1] `http://en.wikipedia.org/w/index.php?title=N-version_programming&oldid=538408306` (visited on 02/21/2014)

functions will deliver wrong results. Because of these restrains *N-Version Programming* is mostly only used in software which controls airplanes, nuclear power plants or medical devices. [L. Chen and Algirdas Avizienis, 1978]

Recovery Blocks

This technique is similar to *N-Version Programming*. In this approach, it also is important to provide 2 or more different approaches of one component of the software. On execution of the program, checkpoints are created before the first version starts. If an error is detected, a different version is executed. The checkpoints are necessary to restore the valid state from, before the error occurred.

One of the main differences to *N-Version Programming* is the fault detection mechanism. The acceptance test, which validates the correctness of the variants can contain various embedded checks and not only an output-check to further improve the detection rate. Most of the time, the first version will deliver the right results. Therefore, the redundant approaches can be slower and more resource consuming alternatives. Depending on the available resources, execution of the component version can even be parallel and not always sequential. [Torres-Pomales et al., 2000]

3.2.4 Android / Mobile phone specific redundancy

Dual SIM On the client side, many mobile phones (especially in China) are dual sim capable. If the main carrier has no network coverage in an area, maybe another carrier has coverage. Sometimes due to software or hardware failures, the network of even major suppliers are unavailable. Of course dual SIM only has a chance to work, if the two used carriers don't share their network.

3.3 Dependency injection

This is a software design pattern, which eases the interchangeability of dependencies. The swapping of the dependency, can happen on compile, as well as on runtime. It is mainly used for changing plugins or for testing purposes by changing the dependencies to mock objects instead of the production ones.

As mentioned in Chapter 2.2.5, it is a good idea to test the monitoring with mock-ups and also to run a debug version for a longer time span. Dependency injection helps in both variants, as the programmer doesn't have to change the code directly, only the provided dependencies have to be switched from the production to mock objects.

The Android client system in Chapter 6.2 from the practical part of this work is already using dependency injection. Dagger was chosen as the preferred injector as it is developed especially for the needs of Android systems.

3.3.1 Dagger

Dagger[2] is a dependency injector for Android and Java with particular attention on speed. It is developed by Square Inc.[3], which provides many open source libraries[4] for Android and other platforms.

To inject a field, *@Inject* from the *javax.inject.Inject* annotations is used. When no producer is implemented, Dagger will use a no-parameter constructor to create the object.

To satisfy dependencies, the *@Provides* annotation is used. The type of the dependency is defined by the return type of the *@Provides* function (see Listing 3.1 for a standard singleton dependency satisfying function).

[2] `http://square.github.io/dagger/` (visited on 03/04/2014)
[3] `https://squareup.com/` (visited on 03/04/2014)
[4] `http://square.github.io/` (visited on 03/04/2014)

```
1 @Provides @Singleton Foo provideFoo() {
2   return new Foo();
3 }
```

Listing 3.1: Provide dependency with Dagger.

Every *@Provides* method must belong to a module. A module is a class with a *@Module* annotation. The classes where the provided dependencies are used, must be registered in the module, else the object graph won't know about them. To use the module and methods, an object graph must be created. An object graph is formed by the *@Inject* and *@Provides* methods. It accepts one or more modules on creation.

Dagger provides many possibilities with its injection:

- *@Singleton* at the dependency satisfying function (see Listing 3.1), causes the function to provide the same instance for all the clients of the object graph.
- Dagger is able to instantiate objects lazily with a *Lazy<T>*. Only at the first call of *get()*, *T* will be instantiated.
- If the type alone is not sufficient to identify the dependency, a qualifier can be created (e.g. a name), which serves as an additional identifier.

3.3.2 Fault injection

The transition from a fault to a failure is well defined in the fault-error-failure cycle [A. Avizienis et al., 2004]. A fault, possibly leads to an error, which is defined as an invalid state of the program. An error possibly leads to further errors in the application. A failure is defined as observable errors at the system boundary.

Faults can be classified according to their persistence and independence.

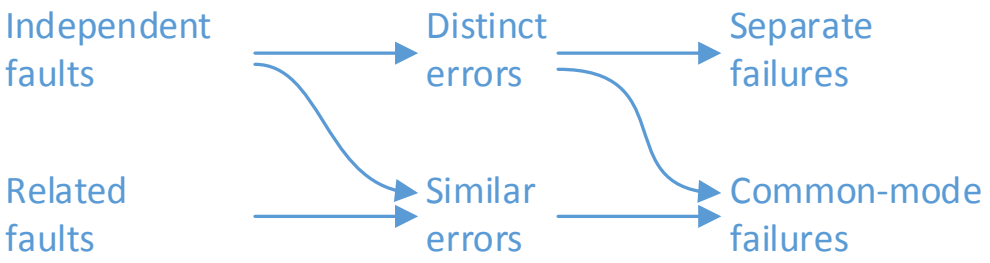

Figure 3.1: Fault classes.

Independence of faults. Faults can be either related or independent. Related faults result from dependencies in the implementation, or from specification errors. A related fault, often leads to a common mode/cause failure. An independent fault is a fault, which is not related and often leads to distinct errors and separate failures. For details, see figure 3.1.

Persistence of faults. Based on the persistence, faults are classified as solid or soft. Soft faults are temporary faults which only need error processing/recovery. After that, the component can be used again. If a component is affected by a solid fault, fault treatment is necessary after error processing. Solid faults are often permanent. [Laprie et al., 1990]

Fault injection is an important part in developing robust software. With this technique, faults are dynamically injected with the goal that the error handling code paths of the software can be verified.

Fault injection can happen on compile time or on runtime. Compile time injection can be further categorized into code insertion, which adds extra code, and replacement of existing code. Runtime injection uses a trigger to inject a fault. These triggers are often time based, where after a specified time, the fault injection is activated automatically, or they could also be interrupt based. An interrupt based trigger listens for a special event like a system broadcast. For testcases, there is often no trigger necessary. The

faulty dependency is injected instead of the production one on each call.

In the scope of the dependability validation process, fault injection can be used as a tool for fault removal or fault forecasting.

Fault removal. During the software development process, fault injection can be used as part of the test strategy in test cases.

Fault forecasting. This approach isn't executed for finding bugs, but for testing the fault tolerance robustness of the software. With fault forecasting, the performance (e.g. coverage, latency) of the fault tolerance mechanism can be estimated. The coverage is the percentage of faults and errors that can be handled. The latency is the time from injecting the fault, to the manifestation of the error or the proper fault handling.

There is also a distinction between fault and error injection. In fault injection, the goal is to 'mutate code', which could eventually lead to an error. Error injection on the other hand, which is sometimes referred as *data-state mutation* [Voas, 1998], changes the state of the program to simulate an already manifested error or failure.

Fault injection with Dagger

Dagger can be used to inject faults. The sample application of Jake Wharton called *u+2020*[5] shows, how to introduce a special debug mode. In this debug build of the application, some functionality can be dynamically changed:

- The network endpoint can be switched to debug, which is a simulated mock server. This mock server delivers around 20 test images. The

[5] `https://github.com/JakeWharton/u2020` (visited on 03/05/2014)

second possibility is the production endpoint, in this case imgur[6], which shows the most viral images of the site.
- With an activated mock endpoint, a delay of loading each picture can be set to simulate the transport time from the server.
- It is also possible to activate an error rate for the loading of a mock picture.
- Other debug information like a pixel grid, the animation speed and picture indicators can be shown.

In the production build of the application, nothing of the debug code will remain in the *.apk* file. This is possible because the debug build overrides a special debug module which is not present in the normal application.

Therefore, dependency injection is also a good solution for the problem of testing long term reliability for the monitoring system, as stated in Chapter 2.2.5. For example, the injection of the web services can be replaced with a wrapper providing this service but which is also able to simulate a failure when needed. The big advantage is that in the normal code nothing has to be changed, but testing is simplified by just overriding the dependencies for the test run, this leads to more structured and easier to understand testing code. [Terasa and Schupp, n.d.]

3.4 Library Tape

Another library of square[7] that helps developers to enhance the reliability of Android and Java applications is *Tape*[8]. *Tape* represents a collection of queue related classes and basically implements a persistent file backed queue.

The library essentially consists of 3 major classes.

[6] `https://imgur.com/` (visited on 03/05/2014)
[7] `https://squareup.com/` (visited on 04/16/2014)
[8] `https://square.github.io/tape/` (visited on 04/16/2014)

- *QueueFile* is a transactional, file-based FIFO with particular focus on speed. Data is written synchronously to the disk. This means that the write operation is completed before the operation returns. The file structure can survive process and system crashes as well as shut downs.
- *ObjectQueue* is the ordering of arbitrary objects. This ordering can be backed by *QueueFile* (on the filesystem) or in memory only.
- *TaskQueue* is an object queue which receives and holds the tasks added by the programmer.

By adding tasks to the queue, Tape is able to save these tasks, as well as the objects which are contained in the tasks, to the file system. This queue is then processed, until every task has finished its execution. In the case of a process or system crash the information necessary to execute the tasks is saved on the file system and thus the application is able to resume the tasks in the queue automatically when it is restarted.

Usually the queue starts a distinct service, where a new thread is started, which allows actions to be completed in the background. When the thread is finished, the function *executeNext()* of the queue is called via the callback. Then the next task is started.

Therefore, *Tape* is a useful tool to execute very important and essential tasks which are not allowed to get lost under any circumstances.

4 Practical Work on the Effective Control System

The initial step of the practical part of this work was the implementation of a *Wirksamen Kontroll-System (WKS)* (Effective Control System) for occupational safety on construction sites. This Android application was implemented in 2013 for Norbert Rabl Ziviltechniker GmbH, which is a civil engineering bureau for fire prevention, occupational health and safety and sustainable building (Green/Blue Building).

4.1 Overview

Responsible supervisors are bound to control and evaluate the implementation of occupational safety on many construction sites every day. On those sites, the supervisors need to fill out a form as well as document the realized safety measures with photos. The WKS system, instead of using paper, allows the workers on the site to document the safety measures with a smartphone. The system allows to send texts, photos and recorded audio. These documentations are then sent to the supervisor which controls them remotely. See Figure 4.1 for the initial concept of the WKS system.

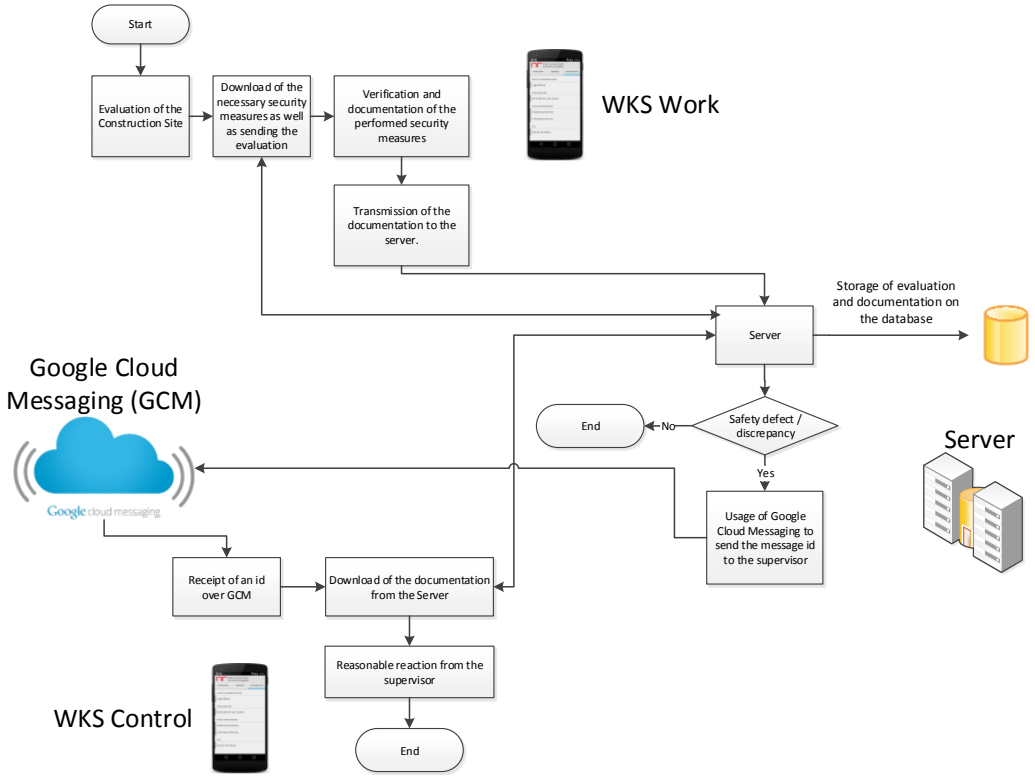

Figure 4.1: WKS Concept.

4.2 Reliability in the WKS system

The reliability is a key factor in a system, which documents the occupational health and safety on construction sites. No data is allowed to be lost ever, as this data could get a key position in upcoming lawsuits. It is also highly important that messages are delivered to the recipient as fast as possible, as otherwise the supervisor cannot attend to his duties.

4.2.1 What is done

Messages are sent and received in a tape queue (see Chapter 3.4). This means that it is highly unlikely to lose a message. The device will try to send a message until it is done. Neither closing the application, nor restarting the device will endanger the data in the queue. There is already a very powerful *DebugServer* present in the application, which implements the Server interface. This is a class, which mocks real Server interaction. With the help of this *DebugServer*, it is possible to test the functionality without accessing a real server, as well as demonstrate the application in areas without network reception.

4.2.2 Future

As of September 2014 there is no finished Server implementation available. Therefore, no service monitors are integrated into the Android application. When a server is available, the application uses GCM and the server for the communication and transmission of messages. Therefore, in the future, especially GCM should be monitored as the only essential external service the application uses. If the service fails, the application needs to switch to polling messages instead of waiting for GCM broadcasts. Another possibility is to use SMS notifications. With an SMS receiver, GCM could be circumvented. Data SMS makes it possible to send SMS messages exclu-

sively to applications, listening to special ports (see Chapter 6.2.3 for more information).

The *DebugServer* is integrated into the main source code of the application. *Dagger* (see Chapter 3.3.1) is already in use but only to deliver singleton instances like Gson or observers. It is possible to make a *Dagger* module, which isn't available in the release flavor of the application. Then, the application can inject different *DebugServer* implementations, which can simulate a range of errors and events on runtime. With this, testing different situations can be done without too much overhead. *Dagger* also implements lazy loading of objects when they are needed. As long as *Dagger* is used, it is assured to deliver valid instances in contrast to statically stored instances which can be affected from the Android life cycle (see Chapter 1.2). A *DebugActivity* is already introduced in the application which is only present in the debug build. This is done with the help of gradle build flavors.

The application is only ever tested on Nexus and older Samsung devices. But the Android ecosystem is fragmented into 5 major versions as well as many different front ends like *Samsung TouchWiz* or *HTC Sense*. Also screen sizes and form factors are highly diverse on the hundreds of available devices on the market. Therefore, a test suit should contain at least the devices and Android versions with the greatest impact on the market share.

5 Service Monitoring Library

The monitoring library with the name *ServiceMonitoring* is free available and located on *GitHub*[1]. The library is licensed under the *Apache License Version 2.0*[2]. The goal of this library is to relieve developers of implementing distinct monitor handlers for all the services and parts of their software they want to analyze. Furthermore it doesn't use any libraries itself and the minimum SDK version is 3 (Android 1.5 Cupcake), which was released in 2009. Therefore, this library will work with nearly every Android application and on 99% of all available devices. According to Google, in August 2013, the market share of devices running a version older than Android 2.2 was about 1%[3].

The *ServiceMonitoring* library takes care of the implementation details of some major questions a developer has to ask himself before starting to work on a monitor. Some of these questions are:

- How does the monitor start when no application process is currently running?
- How can you be sure to have access to the application process at every time while running this monitor?
- How does this monitor shut down if I want it to?

All these issues are already solved and implemented in this library. The developer has to implement and provide the actual monitoring function (observer, analyzer, as well as the event handler), which is then executed

[1] `https://github.com/markini/smonitoring_lib` (visited on 12/28/2013)

[2] `https://www.apache.org/licenses/LICENSE-2.0.html` (visited on 12/28/2013)

[3] `https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net` (visited on 05/06/2014)

by the library. The library provides settings for the start intervals, as well as the type of the monitor, which can be configured before executing. That said, this library doesn't monitor anything by itself. It just provides the framework, which executes the monitoring code.

5.1 User stories

This library was created with a few user stories in mind. It was a key factor to use real life examples. To find such stories, only Android developer were asked to participate. Ordinary Java developers, often don't have experience with the unique behavior of the life cycles of Android applications.

Generic client server application monitoring for the developer Sometimes, a monitor can be used to monitor a server or service before the user is using the application to warn the developer that some client (the Android application) has some sort of problem with the service. Then the developer or server administrator can patch the error before the user takes notice.

Often, problems are very difficult to anticipate due to the many different devices and languages the users are using. Therefore, a monitoring system executed by each user makes sense.

Tornado warning system This is an application which uses GCM to receive warnings about tornadoes in the vicinity. This application will occasionally update its own location but will most of the time be shut down by the system. It is important that the service monitoring works correctly, even if the application isn't running. The monitor has to notify the user if the internet connection is not working or the server/GCM are not functioning properly.

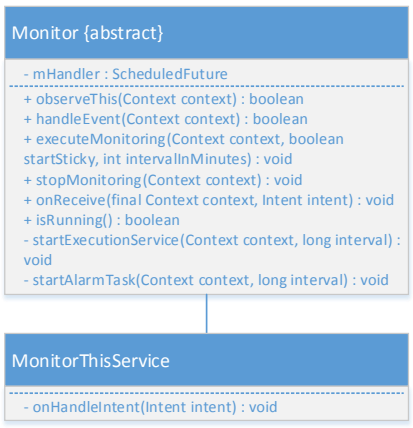

Figure 5.1: Monitor library class diagram.

Implementation of an Android Watchdog Sometimes, monitors are applied to check the state of the application. This kind is often referred to, as a watchdog [Fowler, 2009]. One duty of this monitor is, to reconfirm, if the application is still on and working. Therefore this watchdog monitor must be separated from the application to even survive uncatched runtime exceptions. Then, it is possible for the watchdog to start the application again or to handle other possible problems.

5.2 Structure

ServiceMonitoring is an Android library project ready to be included in any other Android project with API Level bigger or equal 3. It consists essentially of 2 Java files: *Monitor.java* and *MonitorThisService.java* (see Figure 5.1).

To use this library, the developer has to create a new class which extends *Monitor*. After implementing the abstract functions *observeThis* and *handleEvent*, an object of this class can be declared and initialized. To start the monitoring functionality, the client application needs to call the function

executeMonitoring, which is already implemented by the abstract *Monitor* class.

In the next sections, the main functions of this library are explained. The functions *observeThis* and *handleEvent* are abstract and have to be implemented by the application developer. All other functions are ready to be used by the client system by default. Figure 5.2 shows the basic flow of this monitoring library.

5.2.1 Function observeThis

In this function, the client developer has to implement the actual monitoring. E.g. checking the connectivity of the device, or sending a ping to a server/service, needed to run this application. It can also monitor internal states of the application but isn't exactly suited as a classic fault detection monitor (it can only be used as a time-driven runtime monitor, see Chapter 2). It must return false if it detects some kind of malicious behavior, or a failure of the monitored object. If this function returns true, this monitoring instance stops after finishing this function, but will be repeated after the set time interval. If it returns false, the function *handleEvent* is automatically called.

5.2.2 Function handleEvent

As mentioned before, this function gets called if *observeThis* identified a problem. Here, the developer has the opportunity to counter a failing service or inform the user. For example, if a service had failed, the application could switch to another one. It is also possible to send a notification to the user from here, for example via an Android Broadcast.

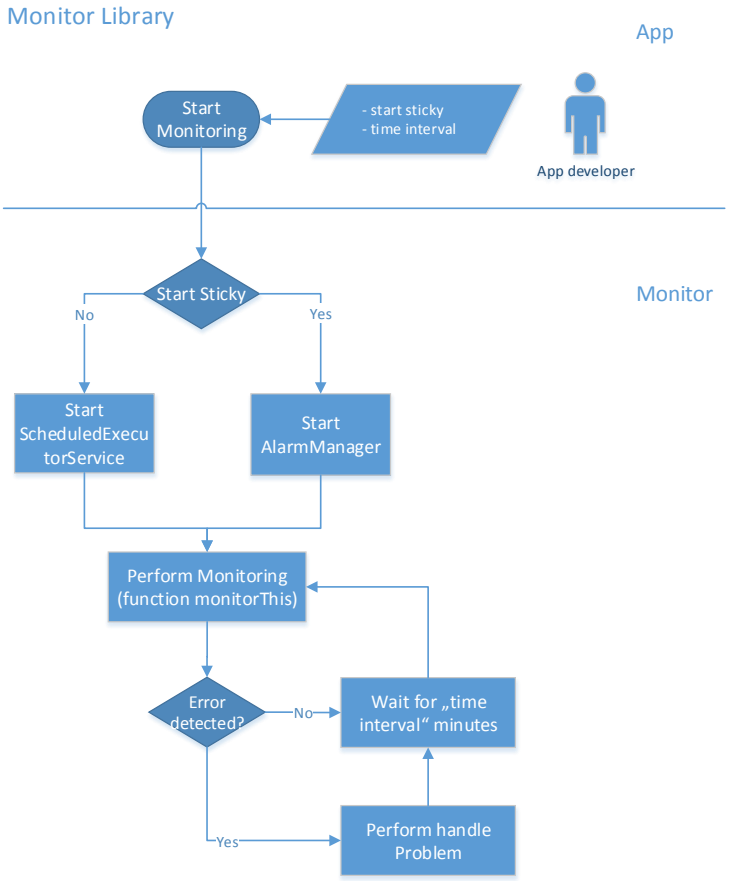

Figure 5.2: Monitoring library flow chart.

5.2.3 Function executeMonitoring

This function starts the monitoring process and takes 3 parameters: The first parameter is an Android *Context*1.4. The second one is a boolean, which indicates if the monitoring should be executed even if the application process was shut down, or if it should stop when the application stops. This boolean flag is called *startSticky*. The third parameter is the interval in minutes, in which the monitor will be executed.

5.2.4 Function stopMonitoring

To stop the monitoring process, the developer has the possibility to call the public function *Monitor.stopMonitoring*. After calling this function, the currently running process of the monitor will continue (if observeThis or handleEvent is executed), until finished but the next instance of the execution will be aborted. *StopMonitoring* deletes the scheduled tasks, which are necessary for the monitoring to start. To restart the process, the developer has to call the function *executeMonitoring* again.

5.2.5 AlarmManager

If the developer decides to set the parameter *startSticky* to true, the function *startAlarmTask* (see Listing 5.1) is called, which uses the Android *AlarmManager* to handle the continuing execution.

```
1 /**
2 * Sets the AlarmManager to start the monitor
3 * continuously in the given interval.
4 * The monitor is started after one second.
5 *
6 * @param context  the application context
7 * @param interval the interval in which the
8 *                 monitor executes (in milliseconds)
```

```
9  */
10 private void startAlarmTask(Context context, long interval) {
11     Intent intent = new Intent(context, this.getClass());
12     PendingIntent pendingIntent = PendingIntent.getBroadcast(
13         context, 0, intent, PendingIntent.FLAG_CANCEL_CURRENT);
14     AlarmManager alarm = (AlarmManager) context.getSystemService(
15         Context.ALARM_SERVICE);
16     alarm.setRepeating(AlarmManager.RTC, System.
17         currentTimeMillis() + 1000, interval, pendingIntent);
18 }
```

Listing 5.1: Set up and start of the AlarmManager.

The *AlarmManager* provides access to the Android system alarm services. These services can send a broadcast to the application at any scheduled time or time interval. The application needs to register a receiver for these broadcasts. Therefore, the Monitor also extends the class *BroadcastReceiver*. Every implementation of the *Monitor* by the client developer, needs to be registered in the Android manifest. This only applies if the Monitor is started with the *startSticky* flag. The broadcasts sent by the system, wouldn't be catched without these receivers. When receiving such a broadcast, the function *onReceive* in the *Monitor* class is automatically called, even if no application process is active at the moment. The passed context to this method is an *ApplicationContext* if the program was already running. If not, only a *ReceiverRestrictedContext* is passed, which doesn't allow certain operations like *bindService* or *registerReceiver*.

The call of *onReceive* runs in the main thread of the application process[4] (if the application was already running, the main thread is the UI thread). This leads to some problems. Running this code on the UI thread would influence the performance of the normal application functionality and users could notice that the application doesn't run smoothly without knowing why. It is even possible that the application isn't responding, and the Android system sends the user a notification whether he wants to shut the whole application down or not. Just creating a new thread like in the *ScheduledExecutorService* approach won't necessarily fix this problem. It is possible for the application

[4] `https://developer.android.com/reference/android/content/BroadcastReceiver.html` (visited on 06/05/2014)

to lose its state as *onReceive* reaches its end. The thread, which is still performing the monitoring code, is then considered stateless.

A single thread, which isn't backed by a working application state, won't be able to access static variables or Android resources. Without Android resources, most of the implementations of *observeThis* and *handleEvent* will probably crash. Another possibility to keep the application process alive, as well as run the code in *onReceive* in the background is to use the call *goAsync()*, which is available since Android API 11. This method returns a *PendingResult* object and the receiver is considered as alive until *PendingResult.finish()* is called. [Vogella[5]]

The solution for this problem in this library however wasn't the usage of *goAsync()*, but to run the code of the *observeThis* function in a distinct Android *Service*, which will keep the application process alive. It has been decided to use an *IntentService*, as this service runs in an asynchronous thread and stops itself when it runs out of work. Normal Android services are not asynchronous, furthermore they won't just stop executing when running out of work. The big drawback in using an *IntentService* is that they won't run concurrent, meaning that the Android system will queue *IntentServices* of the same type, processing them one at a time. See Listing 5.2 on how this service is started, this code is executed in the *onReceive* function. The service itself will then call the function *observeThis*.

```
1 @Override
2 public void onReceive(final Context context, Intent intent) {
3     Intent serviceIntent = new Intent(context.
4         getApplicationContext(), MonitorThisService.class);
5     serviceIntent.putExtra("monitor", this);
6     context.getApplicationContext().startService(serviceIntent);
7 }
```

Listing 5.2: Broadcast onReceive.

Services can't be started by the system, without declaring them in the manifest of the client application. Therefore, every client using this library

[5] http://www.vogella.com/tutorials/AndroidBroadcastReceiver/article.html (visited on 03/22/2014)

with the *startSticky* flag, has to register the *MonitorThisService IntentService* in the Android manifest.

To pass parameters to the execution function of the service, an Android intent is used. To pass custom objects, the class needs to implement *Parcelable* or *Serializable*. For this library, it was decided to use *Parcelable*. Therefore, the application developer has to implement the static field called *CREATOR*. Otherwise, the application will crash on trying to pass the monitor to the service. In this creator field, a new instance of this monitor has to be returned. It wasn't possible to implement this field in the library itself due to the fact that the monitor is abstract. The service won't be able to use the private *ScheduledFuture* member after the conversion to a Parcel. This however, is of no concern, as this member is only used to stop a *ScheduledExecutionProcess*, which is never called by the *Service*.

Sadly, the *Parcelable* approach is one of the most verbose implementations in Android. However, this approach was chosen, due to the easy implementation of this particular *CREATOR* field. Furthermore, *Parcelable* is 10 times faster than *Serializable* and uses less resources[6]. The community is working on solutions to make *Parcelable* easier to use. Examples are the library parceler[7] and the library auto-parcel[8]. With these libraries, it is possible to make a class parcelable without the implementation of the verbose functions *writeToParcel()*, *describeContents()* and the *CREATOR* field. They are however, not suited to be used in a library project, but a good alternative in a standard Android project.

Another possibility to pass the Monitor object to the service, is to use the free library GSon[9]. With this library it is possible to convert the Monitor object to its JSON representation. Then, only a JSON String is passed via the intent. This is however much slower than passing *Parcelables*.

[6] `http://www.developerphil.com/parcelable-vs-serializable/` (visited on 12/28/2013)

[7] `https://github.com/johncarl81/parceler` (visited on 06/06/2014)

[8] `https://github.com/frankiesardo/auto-parcel` (visited on 06/06/2014)

[9] `https://code.google.com/p/google-gson/` (visited on 06/06/2014)

5.2.6 ScheduledExecutorService

If the developer decides to set the parameter *startSticky* to false, the function *startExecutionService* is called, which starts a *ScheduledExecutorService*. In this function, a new thread is created. Without a new thread, the code would run in the UI thread. The only mission of the runnable of this thread, is to call the function *monitorThis*. This function is executed every time the *ScheduledExecutorService* starts the monitoring process. A *ScheduledFuture* is returned after executing the *scheduleAtFixedRate* of the *ScheduledExecutorService*, which is saved as a member (*mHandler*) and is used to stop this monitor in the *stopMonitoring* function (see Listing 5.3).

```
1 ScheduledExecutorService scheduler = Executors.
    newScheduledThreadPool(1);
2 Thread runner = new Thread(new Runnable() {
3     public void run() {
4         if (!observeThis(context)) {
5             handleEvent(context);
6         }
7     }
8 });
9 mHandler = scheduler.scheduleAtFixedRate(runner, 1000, interval,
    MILLISECONDS); //starts in one second
```

Listing 5.3: Starting ScheduledExecutorService.

The *ScheduledExecutorService* runs in the application process. If the application process is shut down by the Android system, none of the scheduled tasks will be executed any more. It would have been possible to start a distinct service which runs the *ScheduledExecutorService* to keep it alive, but for the start sticky case this library is using the Android *AlarmManager*. Furthermore, it isn't necessary to register this service in the manifest.

5.3 Miscellaneous

To summarize what the developer has to do when using this Monitor library:

- Create a class extending *at.marki.ServiceMonitoring.Monitor*.
- Implement the 2 abstract functions *observeThis* and *handleEvent*.
- Implement the *CREATOR* field for the *Parcelable* functionality.
- In the Android manifest: add the new created class as a receiver and register the *MonitorThisService*.

6 Test Environment

In this chapter, an Android client - server environment is introduced, which utilizes the monitoring library (from Chapter 5) to gain an additional layer of fault tolerance in the case of a failing service.

A big percentage of all Android applications on the market are highly dependent on web services. These services are often outsourced to, or provided by a third party. The implementation of new functionality, maintenance and enforcement of reliability falls in the hands of the external parties. With the monitoring of such a Service, the availability and the accuracy can be measured on runtime (see Chapter 2.2 for more information).

One of the main goals of this test environment was to utilize as many Android specific properties as possible, with the goal to differentiate this application from standard Java programs. An Android application is written in Java, but has many unique properties (see Chapter 1).

The test environment is a rudimentary client-server system. It consists of an Android application with the name *MTClient* and a *Jetty* server named *MTServer*. Figure 6.1 provides a basic overview about this client server system.

The Android side of the environment was implemented wit a practical orientation in mind. Therefore, it is functionality-wise very similar to a 'real' application, countlessly found in the *Play Store*[1]. It is able to communicate with different services, which are externally provided, as well as the Jetty server described in Chapter 6.1. It notifies the user about changes like most

[1] `https://play.google.com/store/apps` (visited on 04/08/2014)

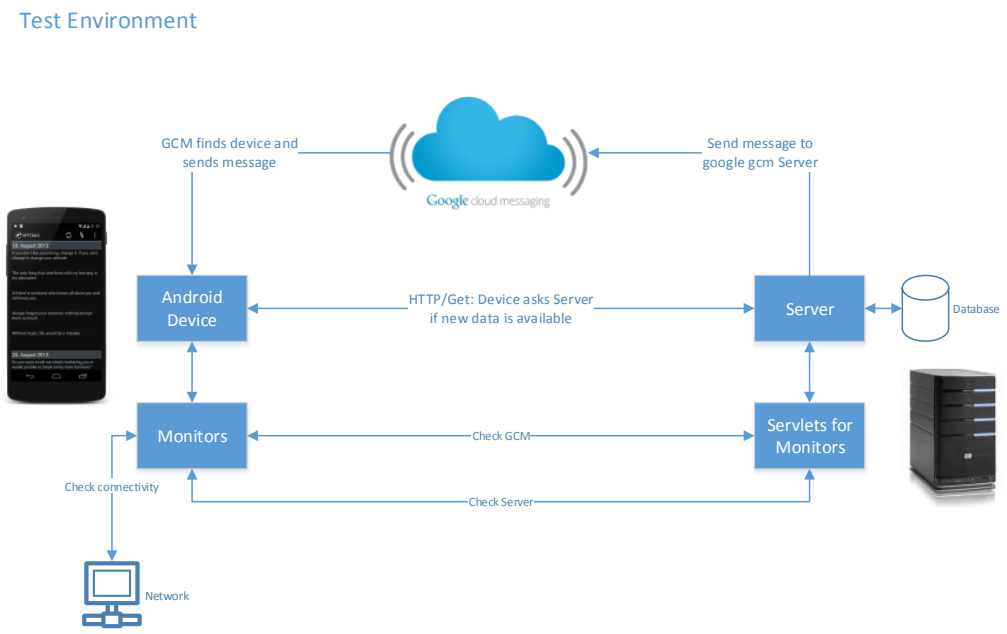

Figure 6.1: Test environment client server overview.

other Android applications, and can incorporate new user content. As this work focuses on Android reliability, the server side of this environment has a lot less functionality, and some parts of it are only mocked.

6.1 Server

The server is a Java application with a Jetty[2] server and a *Swing* graphical user interface (GUI). The messages have to be entered by hand into an input field. There are 2 distinct buttons (see Figure 6.2).

By clicking the button *Set Message*, the message is fetched from the input field. The server initializes a new message object and creates an *UUID* to

[2] `https://www.eclipse.org/jetty/` (visited on 05/28/2014)

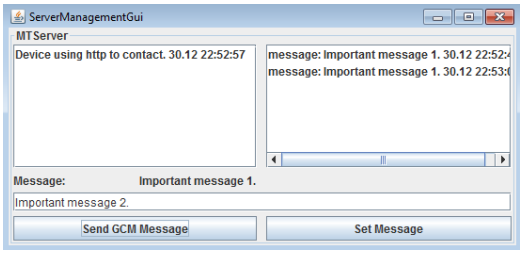

Figure 6.2: Server GUI.

identify this message. The *Send Message* button forwards the currently set message to the registered clients via GCM.

The server is also capable of receiving HTTP requests from the client application. Via HTTP it also can transmit messages to update the client. Although its main purpose is to use GCM to send new messages to the receiving application. To find the client application over GCM, the server needs to know the GCM ID of the Android device, where the client is installed on. The server has the ability to receive and store the GCM IDs of the clients. It is possible to register more than one device on the server and also to send a GCM message automatically to any number of clients/devices. The GUI shows the already sent messages on the right and also if a client requested a message over HTTP on the left (see Figure 6.2).

The server also has to implement servlets to receive requests from the Android applications monitors. These servlets will assert the state of the server and GCM. The implemented servlets are:

- *ServletServerStatus*: This servlet returns the *HttpServletResponse SC_OK* indicating that the server is reachable. Read more about this servlet in the monitor chapter 6.2.5.
- *ServletCheckGCM*: On *doGet*, this servlet sends a GCM message to the requesting *MTClient* application. Read more about this servlet in the monitor chapter 6.2.2
- *ServletRegisterGCMId*: The client can register its GCM ID by sending an HTTP request containing its ID to this servlet.

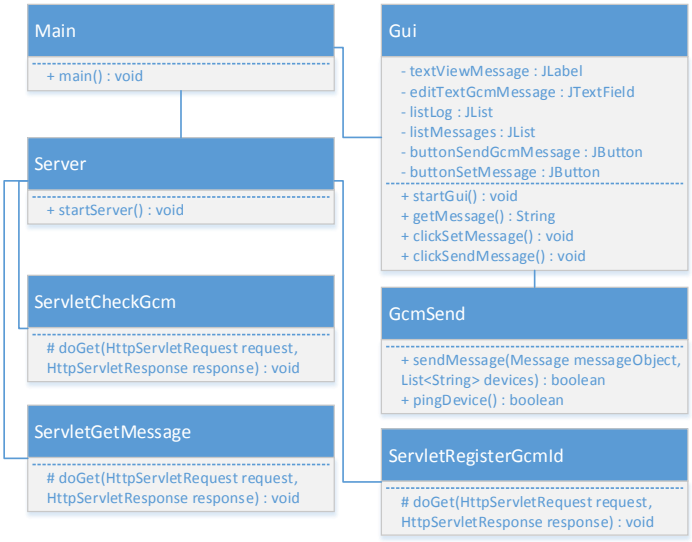

Figure 6.3: Server class diagram.

See Figure 6.3 for the class diagram of the server.

6.2 Client

The mission of the client, is to inform the user that a new message or a warning arrived. The user also has the possibility to instantly view the message. If the client's main activity isn't active, a notification is displayed to indicate a new arrival. If the main activity/fragment is currently shown, the message is just displayed in the *ListView* of the main fragment. But the main purpose of the implementation of this small messaging application, is to show the feasibility of implementing service monitors, as well as redundancy approaches on an Android device. See Figure 6.4 for the class diagram of the client. The next Chapters will explain the GUI as well as the monitoring capabilities of the application.

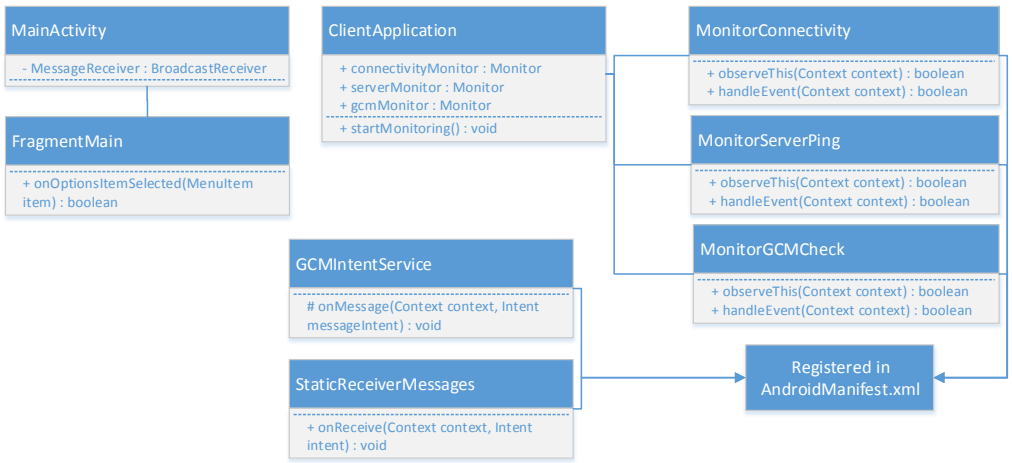

Figure 6.4: Client class diagram.

6.2.1 MTClient UI and capabilities

The client is an Android application with a very simple UI. The main fragment consists essentially only of the *ListView*, which shows the incoming messages. Of course this application utilizes a native Android *ActionBar*, which is part of the activity.

There are no buttons in the fragment. The whole, by the user controllable functionality, is handled by the *ActionBar* (see Figure 6.5 for all action items in the *ActionBar*).

Get Message The *Get Message* action item establishes an HTTP connection to the server and pulls the currently set message from the server, if not already on the device. Normally this would not be necessary as GCM delivers every new message automatically to the application, but this functionality is sometimes very handy in a test environment and can also be used to update the state of the application faster after booting the device. The application also switches to this approach of message acquisition if the GCM service

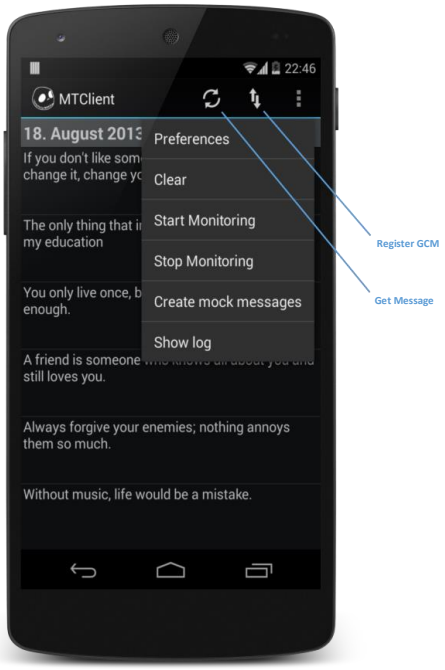

Figure 6.5: MTClient UI with ActionBar items.

encounters a failure (see Chapter 6.2.2).

Register GCM This functionality sends the GCM ID of the device via HTTP-GET to the server.

Preferences The second fragment of this application is the preferences fragment. As of now, only the server IP can be changed in this fragment.

Clear On clicking this action item, all messages are deleted from the *ListView* adapter and the *SQLite* database.

Start Monitoring All 3 monitors are started (Connectivity Monitor 6.2.5, Server Monitor 6.2.5 and GCM Monitor 6.2.5).

Stop Monitoring All 3 monitors are stopped. A currently running monitor will not be cancelled but they will not start anew, until *Start Monitoring* is called again.

Create Mock Messages On clicking this action item, the application generates pre-defined messages for testing purposes.

Show log Each monitor saves its execution results to a *SQLite* database. On pressing the *Show log* button, a new dialog view is opened and the logs are shown in an Android *ListView*. The results can of course be extracted as a *.sql* file for comparing purposes. To realize this functionality, a new object *Log* was created, which holds an unique id, the time (as a *Long* number) and the actual message. A new log entry is created and automatically saved to the database on every execution of a monitor. On every failed monitoring

attempt, all available data which could lead to the reason is saved in the logging message. Also, every successful monitoring attempt is logged as this is an important indicator for the reliability of the monitoring system (see Chapter 6.3.5). For *LogCat* logging, the free library *Timber*[3] is used. Therefore there are no confusions in the source code, whether *LogCat* or *SQLite* monitor logging is intended by the developer.

To log every step of all monitors will slightly influence the performance and the battery. Therefore extensive logging could be a special feature in a specialized version and not activated by default in the productive system. To make the switch between a specialized version with a complex logging system, which is only used by a handful of test users, and the productive version easier, dependency injection (see Chapter 3.3) could be used. Then the productive version would not suffer from the performance loss and energy usage of the extensive logging system.

6.2.2 GCM Capability

As already stated, it is mandatory for the client to receive messages even if the application process isn't running. To receive *Google Cloud Messaging* messages, a *GCMBaseIntentService* named *GCMIntentService* is implemented. This service is also registered in the *Android Manifest* (see Listing 6.1). It behaves a lot like a static *BroadcastReceiver* (see Chapter 1.4). The *action* states, what this receiver does, in this case it will receive GCM messages. To be able to receive those messages, the *c2dm* permission *SEND* is needed. Otherwise the messages would be ignored by this receiver. There is no explicit connection between the implementation of the *GCMIntentService* and the static receiver. The system will find it automatically because of the stated category, in this case the *'at.marki.Client.receiver'* package.

```
1 <!-- GCM START -->
2 <receiver
3     android:name="com.google.android.gcm.GCMBroadcastReceiver"
4     android:permission="com.google.android.c2dm.permission.SEND">
```

[3] `https://github.com/JakeWharton/timber` (visited on 04/02/2014)

```
5       <intent-filter>
6           <action android:name="com.google.android.c2dm.intent.
                RECEIVE"/>
7           <category android:name="at.marki.Client.receiver"/>
8       </intent-filter>
9   </receiver>
10  <!-- GCM END -->
```

Listing 6.1: GCM BroadcastReceiver.

When receiving a new GCM message from the Server, the function *onMessage* of the class *GCMIntentService* is automatically invoked. This function is responsible to parse the incoming GCM message and to store it on the *SQLite* database. Furthermore it will inform the user about the new message (more in Chapter 6.2.2). The receiver is registered in the manifest, therefore it is a static receiver. A static receiver is triggered even if the application isn't running at the moment. A dynamic receiver, which is mostly instantiated in the activity or the application class would not survive the closure of the application process.

GCM Registration

To be able to receive GCM messages, the application has to register the device at the Google GCM server (Listing 6.2). This has to be done at least once for the device. Best practice, is to check if the device is registered at every start of the application. This approach was deprecated by Google while writing this application. Now GCM is part of the Google Play Services[4].

```
1  void manageGCM() {
2      GCMRegistrar.checkDevice(this);
3      GCMRegistrar.checkManifest(this);
4      final String regId = GCMRegistrar.getRegistrationId(this);
5      if (regId.equals("") || !GCMRegistrar.isRegistered(this)) {
6          GCMRegistrar.register(this, GCMIntentService.SENDER_ID);
7          Timber.d("received ID: " + GCMRegistrar.getRegistrationId(
               this));
```

[4] `https://developer.android.com/google/gcm/index.html` (visited on 12/30/2013)

```
8       } else {
9           GCMRegistrar.setRegisteredOnServer(this, true);
10      }
11  }
```

Listing 6.2: Manage GCM.

Download Messages via HTTP-GET Problems

The application is also able to download messages via HTTP. An error, which appears in many applications is to lose the application state, while downloading very big chunks of data over a slow connection. Most downloading tasks on an Android system are executed in an *AsyncTask* or just a new thread. In most cases, this is a sufficient approach but it can lead to unforeseen results. As stated before, this application stores its messages in an *SQLite* database. If the user decides to close the application, while the saving operation is still executed, it is possible to save a corrupt message to the database. Therefore this application uses a Service which starts the *AsyncTask*. This service will not be shut down by the system until the *AsyncTask* has finished its execution, or the system is on critical low memory. Another approach would be to use the library tape from square - coupled with a Service and a Task (see Chapter 3.4).

To send requests, the library HTTP-request[5] is used. This is a convenience library for using a *HttpURLConnection*. Listing 6.3 shows how HTTP-request contacts the server and asks for the currently set message. The server returns a *JSONObject* which contains the message string, as well as the id.

```
1 HttpRequest request = HttpRequest.get(url).connectTimeout(30000).
      readTimeout(30000);
2 if (request.ok()) {
3     JSONObject jsonObject = new JSONObject(request.body());
4     String messageString = jsonObject.getString("message");
5     String messageId = jsonObject.getString("messageId");
```

[5] https://github.com/kevinsawicki/http-request (visited on 12/30/2013)

```
6       Message message = new Message(messageId, messageString, System
            .currentTimeMillis());
7       Data.addMessage(context, message); //saves message on the
            database
8   }
```

Listing 6.3: HTTP Request.

Handle GCM Receive

As the invocation of *onMessage* is called regardless of the state of the application, it is important to handle the different possible states. If the application was already running and the activity is active, the adapter of the main fragment has to be updated. Then the user instantly sees the new message. If the activity wasn't active (active means visible on the screen) at the moment, the user must be informed with an Android notification. This notification shows the application name, informs the user that a new message has arrived and opens the activity when the user clicks on it. The notification also makes a sound, if the current sound profile allows it.

To achieve this behavior, the *GCMIntentService* function *onMessage* sends a custom ordered broadcast to the application. Furthermore there are 2 *BroadcastReceiver* implemented, which can receive this broadcast. One of them is static and has to be registered in the *AndroidManifest* (Listing 6.4). The other one is dynamic and is registered in the overwritten *onResume* function (Listing 6.4) and unregistered in the overwritten *onPause* function of the main activity. The intent filter (the action element in the static receiver), declares the broadcasts the receivers will be able to catch.

It is important to assign a higher priority to the the dynamic receiver. In this case it received a priority of *30*. The static receiver a lower priority of *20*. If the activity is shown and the receiver receives a broadcast, it can update the main fragment and abort the broadcast. Then the static receiver will not be informed. If the main activity isn't shown, only the static receiver receives the broadcast and starts the Android notification to inform the user about the new message.

```
<receiver
    android:name="at.marki.Client.receiver.StaticReceiverMessages"
    android:enabled="true"
    android:exported="false"
    <intent-filter android:priority="20">
        <action android:name="intent.filter.marki.message.receiver
            "/>
         </intent-filter>
</receiver>
```

Listing 6.4: Registration of static BroadcastReceiver in the manifest.

```
//filter for normal messages (gcm receiver)
IntentFilter filterMessage = new IntentFilter(getString(R.string.
    intent_filter_message_receive));
filterMessage.setPriority(30);
this.registerReceiver(messageReceiver, filterMessage);
```

Listing 6.5: Registration of dynamic BroadcastReceiver in the main activity.

6.2.3 SMS Capability

Besides getting messages via Google Cloud Messaging and HTTP, the client application is able to receive messages via *SMS*. Of course this is only possible for devices with a SIM card and a working contract with a mobile network provider.

Sending SMS messages

To enable this approach, the client has to transmit its phone number to the server. It is impossible to do this via the HTTP GCM registration as the phone number is not physically stored on all SIM cards or broadcasted via the network. Even if the phone is able to read the number with the permission *READ_PHONE_STATE* it is possible to get the number of a previously installed SIM card or null.

Therefore, the registration of the phone number has to be done by sending a SMS to the server. The client handles this in the function *registerWithSms* (see Listing 6.6). The Android *telephony.SmsManager* enables the application to send a SMS to a specified receiver, which is in our case the phone number of the Server, which is stored in the global strings resources. For this code to work, the permission *SEND_SMS* has to be set in the Android manifest. For practical reasons, the message of this SMS also contains the GCM ID. The server has to parse this ID, as well as the phone number of the sender. After that, all information needed to send SMS and GCM messages to the MTClient application are gathered.

The *PendingIntent sendingPendingIntent* and *deliveringPendingIntent* are coped with 2 broadcast receivers set in the functions *registerSentReceiver* and *registerDeliveringReceiver*. Those 2 receivers are registered in the *MainActivity* and get unregistered on leaving the activity, which happens when the user closes the application. The *sentReceiver* gets informed by the System if:

- SMS sending was successful.
- A generic failure happened.
- Device was unable to send SMS due to no service availability.
- No pdu was provided.
- Failed because the radio was explicitly turned off.

The *deliveringReceiver* gets informed by the System if the delivery was successful or if the operation was cancelled.

```
1  private void registerWithSms() {
2      PendingIntent sendingPendingIntent = registerSentReceiver();
3      PendingIntent deliveringPendingIntent =
           registerDeliveringReceiver();
4
5      GCMRegistrar.setRegisteredOnServer(getActivity(), true);
6      String gcmId = GCMRegistrar.getRegistrationId(getActivity());
7
8      SmsManager sms = SmsManager.getDefault();
9      sms.sendTextMessage(getString(R.string.server_sms_number),
           null, "MTClient GCM:" + gcmId, sendingPendingIntent,
           deliveringPendingIntent);
10 }
```

Listing 6.6: Register with SMS function.

Receiving SMS messages

If the registration of the phone number on the Server was successful, the application is able to receive SMS messages. For this, a *BroadcastReceiver* is necessary. This receiver called *SmsReceiver* must be registered in the Android manifest (see Listing 6.7). Receiving SMS also needs the permission *RECEIVE_SMS*.

```
<receiver
    android:name=".receiver.SmsReceiver"
    android:permission="android.permission.BROADCAST_SMS">
    <intent-filter android:priority="2147483647">
        <action android:name="android.provider.Telephony.
            SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

Listing 6.7: Registration of SMS BroadcastReceiver in the manifest.

As the *BroadcastReceiver* receives all incoming SMS, regardless of primary destination, it is important for the Server to tag the message with an identifier. It was decided to insert the tag *'at.marki'* in the body of the SMS message. The *BroadcastReceiver* has the mission to parse all incoming messages and look for this tag. If this tag is not part of the body of the SMS, it has to ignore it and let the broadcast continue. These messages will then be received by the (other) SMS applications installed on the device.

Intercepting SMS

It is important to intercept all incoming SMS messages, which are destined for this client. That means that SMS messages tagged with *'at.marki'* will not

be shown in other SMS applications on the device. For this to work, some preconditions must be set.

As Listing 6.7 shows, the priority of the SMS receiver is *2147483647*, which is the value of maximum Integer. No other application on the device can have a higher priority. The standard Android SMS application on devices with stock Android has the priority 0. After receiving a message and confirming that it is a message sent from the test environment server, the broadcast can be aborted with the call *BroadcastReceiver.abortBroadcast*. A problem arises, if there is another custom SMS application, with the same maximum priority installed on the device. Then it is uncertain which application receives the broadcast first. If the other application receives it first and aborts the broadcast, the *MTClient* has no chance to receive and parse this message.

In Android *KitKat* however (which was introduced after the implementation of the code above), the SMS API changed. There, the concept of a default SMS application was introduced:

- Only one application can be set as the default application and the user has to set this in a dialog.
- Only the default application can write to the SMS Provider.
- The default application receives every SMS. Only the default application is able to abort SMS broadcast.

Therefore the only possibility to intercept standard SMS messages on *KitKat* is to ask the user if he wants to set the *MTClient* as the default SMS application. The *MTClient* is not predestined to be a SMS application, it would however, send all untagged messages to the real SMS applications on the device. This would mean that the *MTClient* just forwards the majority of the messages.

Intercepting Data SMS

Another possibility to circumvent the security feature which came with the default SMS application on *KitKat*, is to use data SMS. Data SMS are a special-

ized form of messages which contain a data body instead of a text message. These messages aren't catched by the default SMS application. Data SMS are sent to a specific port. To receive such messages, the developer has to add the action *<action android:name="android.intent.action.DATA_SMS_RECEIVED" />* with an arbitrary port *<data android:port="1234" />* to the static SMS receiver in the *AndroidManifest*.

Data SMS messages are only catched by the BroadcastReceiver listening to this specific port. A problem could occur, if another application is listening to this port for data SMS messages. Then, it isn't certain which application receives the message first and if the second application receives it at all.

To send such a message, the telephone number and the port (in this case '1234') have to be passed as parameter to the *SMSManager sendDataMessage* function. Furthermore, the data itself is passed as a byte array, instead of a String like in the *sendMessage* function.

6.2.4 Application Shut Down

As described, the receiving of a new message must work even if the application is terminated by the Android system or by the user itself. When the user leaves the application via the back button, the application process is shut down by the System after approximately 30 to 60 minutes (on a Nexus 4 with KitKat). The user can also close it instantly via the 'running services' screen.

A third possibility to close the application is to use the *force close* button in the Android settings. This way of closing the application is special in a very negative way. After force closing an application, all static receivers are unregistered from the Android system. Receiving a GCM message or any other static event implemented in the manifest file of this application won't be triggered anymore, until the application is manually started by the user.

6.2.5 Monitoring

There are multiple monitors in use for this application. Of course the already introduced Monitor library is utilized. As this application needs to receive messages even if currently not active, all monitors need to start even if the application is no longer running, so the monitors need to be started with the parameter *startSticky* set to true.

Monitor Internet

The first monitor checks, if the device is in some way connected to the internet. If no internet connection is available, there are not many options for the application to fulfill its tasks. For this functionality the *observeThis* function of the *MonitorConnectivity* monitor class, checks if connectivity is available with the provided APIs of the Android system (see Listing 6.8).

```
1 ConnectivityManager cm = (ConnectivityManager) context.
      getSystemService(Context.CONNECTIVITY_SERVICE);
2 NetworkInfo netInfo = cm.getActiveNetworkInfo();
3 return netInfo != null && netInfo.isConnectedOrConnecting();
```

Listing 6.8: Call of ConnectivityManager.

If no connectivity is available, the *handleEvent* function will halt all further monitoring events, until the next time the connectivity monitor is called as no other monitor will succeed without connectivity.

Internet Monitoring Tweak To tweak the internet monitoring ability, a special broadcast receiver for connectivity could be implemented in the Android manifest (see Listing 6.9).

```
1 <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
```

Listing 6.9: Connectivity static receiver.

The Android *ConnectivityManager* of the system will call the *onReceive* function of this receiver every time the connectivity status changes. If no internet connection could be established, it is favorable to suspend all scheduled service monitoring tasks to save battery life. After receiving the next connectivity change broadcast by the system, the monitoring, as well as the application functionality can be resumed. This little tweak would not improve the monitoring itself but help making the application more energy efficient.

Switch to SMS It was planned and already implemented that if no internet connection is possible, but the device still has access to the GSM network, to switch the functionality of receiving messages to the Short Message Service (see Chapter 6.2.3). However, this was abandoned due to the changes of the SMS API in KitKat.

Monitor Server

The second monitor verifies, if a connection to the server is possible. This is implemented with a simple HTTP request in the *observeThis* method of the overwritten monitor class. This check confirms, if the HTTP requests returns the state *OK*. In a productive implementation, a complete check of the server capabilities could be implemented, which is here not necessary, as a great part of the server is only mocked. A way to implement a better solution, would be to let the server test itself every 30 minutes and return the result of the last check in this HTTP request.

If the check is successful, the next monitor is started. If it isn't successful, it is indicated that the server is experiencing issues of some sort. In the *handleEvent* method, the application automatically sends a SMS and a mail to the server administrator. In a real world application with thousands of installed applications this wouldn't be the best solution, but it is an ok fault handling for this test environment.

Monitor GCM

The implementation of this monitor is a little bit tricky, due to the issue that IP-ranges of *android.googleapis.com* are not published by Google. It is possible to use *nslookup*[6] on *android.googleapis.com* to learn the IP address, but this approach provides mostly unusable results. As directly pinging the GCM servers isn't of much use, the application implements a more complex but in the end more useful approach of a GCM monitor.

The idea is to send a request to the server via HTTP-GET, which forces it to send a GCM message to the client application. This pre-defined message is catched in the *GCMIntentService* and a *receivedPing* flag is set. After a waiting period of approximately 10 seconds the monitor checks if the *receivedPing* flag is set and returns the result. If the flag wasn't set, the monitor waits for another 30 to 120 seconds. If it still didn't receive the GCM message the method *handleEvent* of the monitor is called.

If GCM isn't working properly but the *MTServer* is, the application needs to switch to HTTP-GET polling. The interval is set to every 5 minutes. If on the next monitor run the application detects that GCM is working again, the HTTP polling will be deactivated (see Chapter 6.2.2 for further information regarding HTTP-GET).

Run-order and event handler

If the internet monitoring fails, it isn't necessary to run the server and GCM monitoring. Therefore the monitors are chained to each other and won't run synchronously. The first monitor to run is the internet monitor. The second one the server monitor, and the last one the GCM monitor.

The *MTClient* application has another event handler (*EventHandler.class*) in addition to the *handleEvent* function which is implemented in each monitor. This is necessary because one monitor has not enough information about

[6] `https://en.wikipedia.org/wiki/Nslookup` (visited on 12/31/2013)

the current state of the services to decide the further actions on his own. The event handler collects all available information of the monitors and enables one of the following states:

- Default (GCM)
- HTTP
- SMS

The states are set and re calculated in the synchronized function *calculateApplicationEvent* after every execution of the functions *handleEvent* and *observeThis* of every running monitor. The flags which are set by the monitors to true or false are:

- Connection
- Server
- GCM

These values, as well as the application states, are stored in the Android *SharedPreferences*, on the internal storage of the system. They cannot be directly stored in the event handler, as the stack is too volatile in an application which can be shut down by the system at any time. The state handler is called when a monitor has finished its job and already set the value to the flag mentioned above. This process is displayed in Figure 6.6.

To activate the state *GCM* or *SMS*, no further actions are necessary because the server sends a message to the device and the GCM and SMS broadcasts are automatically catched by the application, whether the client is running or shut down. To activate the state HTTP-polling however, the client has to set a scheduler similar to the monitoring schedulers to start the message download (see Chapter 6.2.2).

To decide the time interval for the execution of each monitor is a rather complex task. Choosing a real short interval would result in polling the server and keeping the application alive indefinitely. This would result in a high energy consumption and render the use of energy saving GCM useless.

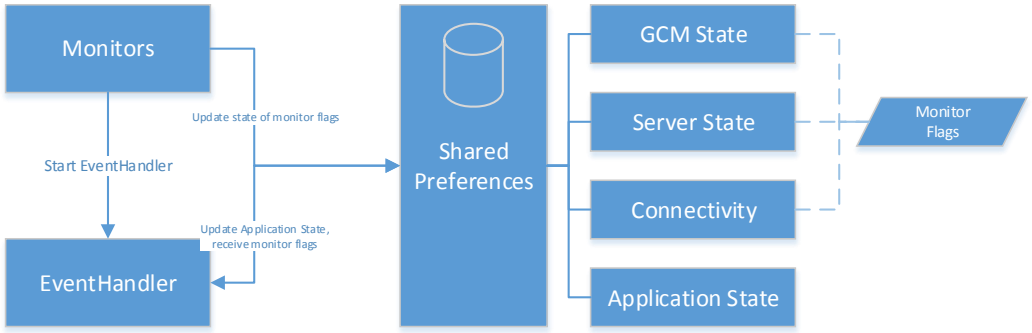

Figure 6.6: Monitor state handling.

It is possible that the same message arrives more than once because of the complex monitor and redundancy architecture of this test client. Therefore every message on the server is constructed with a random UUID[7]. Even if the same message arrives multiple times it will only be displayed once.

As compatibility with older Android devices is of no concern, the minimum API level of the client application is 18.

6.3 Monitoring Results

6.3.1 General Results

Most of the time, the 3 distinct monitors behaved like expected. The Android *AlarmManager* started the monitors even if the application process was long closed by the system. The scheduled events in the *AlarmManager* are not even interrupted by shutting the device down and restarting it. Of course the monitor won't wake the device up when shut down. Only a few distinct applications like the alarm clock are expected to work even if the device is

[7] `https://en.wikipedia.org/wiki/Uuid` (visited on 12/31/2013)

shut down. Therefore this was a design decision that the monitors won't be triggered in a sleeping state. However, the monitor library could be refitted if necessary when using the sticky mode to wake the phone up to start the monitoring.

6.3.2 Monitor Connectivity

To determine the connectivity with the *ConnectivityManager* of the Android API is simple, precise, fast and worked without any issues. The *ConnectivityManager* knows, if the phone has access to the internet via the mobile network, *WIFI* or *Bluetooth*.

6.3.3 Monitor Server

Even with a slow internet connection, an HTTP request to the server with a very small payload is generally processed very fast. The only duty of the server is to set the response to 'OK' to let the client know that he is still accessible. This doesn't however show if the server is in a malicious state.

Ideally, also the server has a running monitor which detects problems in the server environment. For example, if the database server is offline, the implemented HTTP request would still succeed. A real message call however would fail. A monitor, implemented by the server should continuously assert the database and set the state somewhere reachable for the check servlet. If the server is in a malicious state, it will set the response for the client to 'Not OK' or a more specific flag. If the server already knows that he is defect, the client doesn't have anything to do but try again later or switch to a backup server.

6.3.4 Monitor GCM

As already explained, the IP-ranges of *android.googleapis.com* are not published by Google, but the service is usually accessible on port 5228. It is very hard to determine if GCM is working without actually sending a message. Therefore the approach was to send a message from the server to the client via GCM. Most of the time this worked without any issues. The messages arrive in about 2 to 20 seconds and the state of the client can be set to 'OK'.

Sometimes however, especially when using the *WIFI* connection on a router, there might occur problems with GCM. After approximately 5 minutes after receiving the last GCM message, new messages are not received instantaneously. Only after about 15 minutes, all missing messages in between this time frame are received at once. This is not the fault of GCM, rather it is an over-zealous router in between the client and the internet.

On the Android side, there is no indication that the connection failed. Therefore, it is assumed that neither Android nor the Google servers terminated the GCM connection. After 15 minutes after receiving the last successful message however, Android sends out a cooee[8] packet to tell Google to keep this connection alive. This call will not go through to the Google servers and after a number of several retries a new connection to GCM will be set up. This new connection will lead to receiving all the missing messages of the last minutes. Therefore, there is a 10 minute window, where the application can't receive messages.

It isn't completely clear why this is happening, but a theory is that the router deletes the mapping of the Google side port to port 5228 after around 5 minutes of inactivity. Therefore, there is no route from the Google server to the client. The cooee packet implies a keep alive strategy from Google especially for cases like this one.

This is a major problem not only for the application functionality itself but

[8] `https://en.wikipedia.org/wiki/Cooee` (visited on 12/29/2013)

also for the monitor. The monitor will warn that GCM is no longer running, which is essentially correct.

Other Android applications which are using GCM (like Google Hangouts), will most likely send out a ping from the GCM server every 2 or 3 minutes. Another possibility would be to forcefully reset the Android connection to GCM after 5 minutes of inactivity, when connected via *WIFI*. These solutions however, will lead to a higher battery drainage.

There are further issues with the GCM service, a few of them aren't documented by Google.

The device needs to be re-registered if the version of the application changes (this is now documented). It also needs to be re-registered if the Android version has been updated.

The GCM registration also often fails with an IOExeption containing the flag *SERVICE_NOT_AVAILABLE*. As it is not exactly determinable why this happens. The best countermeasure is to keep trying to register the application with an exponential backoff.

On some devices, the GCM registration ID is created but it will fail with an Exception regardless. If this happens, the ID can be catched in the *GCMBroadcastReceiver*, if the *BroadcastReceiver* in the *AndroidManifest* contains the action *<action android:name="com.google.android.c2dm.intent.REGISTRATION" />*.

6.3.5 Reliability

As stated in the Chapters 2.2.5 and 3.3 there are multiple possibilities to verify the long time reliability of a monitoring system. One of the most important factors was, to test it over a longer time period on an actually used device. Such a device must comply with a set of defined conditions:

- The device must be used every day or nearly every day (a weekend without usage is also a good test case).
- Many different applications should be used.
- The user makes phone calls.
- The device should sometimes be turned off or switched to airplane mode.

Therefore a modified client application was handed over to a few users. This application saves a log about the monitors and the corresponding event handling. The log describes if and when a monitor was started, if and when an event handler was started and if and when the device was shut down. To log the start and the shutdown of a device, the *BroadcastReceiver Intent.ACTION_BOOT_COMPLETED* and *Intent.ACTION_SHUTDOWN* must be implemented and registered in the *AndroidManifest*. The logs showed that a monitor, which is coupled with the Android *AlarmManager* is very hard to kill. It successfully survived long time usage with many device shut downs / restarts.

As the *AlarmManager* isn't part of the application, it can survive uncatched runtime exceptions of said application. Therefore, it is possible to implement a watchdog (see user stories in Chapter 5.1), based on an *AlarmManager* action. This watchdog however, will lag after the exception, as it will only check the system based on the pre configures time frame.

6.3.6 Conclusion

The testing phase revealed that services, which are not under control and maintenance of the Android application developer, can be relatively reliable monitored. Though, the developers have to consider multiple little issues which are often unique in a smart phone environment. If a failure in the behavior of a service is found, there are different approaches the application can take. The simplest approach, is to select a backup service. Furthermore, most devices offer a big selection of hardware components which enables the developer to choose between multiple options to respond to the failure.

During the test phase of the service monitoring, it came to light that the Android systems already sends information about the the most important local (on device) occurrences via public broadcasts (See Chapter 1.4). Therefore, targeting 'on device services' with the monitor library is often not necessary. If a system broadcast exists, a *BroadcastReceiver* can be registered. Google is constantly improving the monitoring system of the Android system. More broadcasts for different events are getting implemented as the system matures. The documentation however for this part is still lacking.

Most of the problems which were encountered by the monitoring, where found on the GCM part (see Chapter 6.3.4). The monitoring of the GCM service was also the initial catalyst for the implementation of a library, as this service seemed to be unreliable at times. It came to light however that most of the problems with GCM weren't actually due to service failures, but rather to problems with some *WIFI* routers and *GCM* registration failures. It doesn't help that most of these failures are undocumented on the Android development site. The log of the monitor helped to identify the reasons why no connection to *GCM* was possible, but such logs should exist in every Android application in the testing phase. Key parameter in the log files were the connection type, the Android version and the device manufacturer.

Appendix

Acronyms

AOT ahead-of-time. 5
API application programming interface. 1, 2, 9, 26, 27, 47, 52, 70, 72, 76, 77
ART Android Runtime. 4, 5

CPU central processing unit. 2, 27

GCM Google Cloud Messaging. 10, 25, 26, 43, 46, 58, 60, 62–64, 67, 68, 71, 74, 75, 78, 79
GUI graphical user interface. 57–59

JIT just-in-time. 2, 4, 5

MOP Monitoring-Oriented Programming. 15, 16

SLA service level agreement. 20

TMR triple modular redundancy. 32

UDDI Universal Description, Discovery and Integration. 18
UI user interface. 8, 51, 54, 60

VM virtual machine. 2

WKS Wirksamen Kontroll-System. 41
WSDL Web Services Description Language. 18

Listings

Bibliography

Albari, Mohamed Ziad (2005). "A taxonomy of runtime software Monitoring systems." In: URL: http://www.informatik.uni-kiel.de/~wg/Lehre/Seminar-SS05/Mohamed_Ziad_Albari/vortrag.pdf (visited on 12/08/2013) (cit. on pp. 11, 28).

Ameller, David and Xavier Franch (2008). "Service level agreement monitor (SALMon)." In: *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*. IEEE, pp. 224–227 (cit. on p. 18).

Arzt, Steven et al. (n.d.). "How useful are existing monitoring languages for securing Android apps?" In: (cit. on p. 2).

Avizienis, A. et al. (Jan. 2004). "Basic concepts and taxonomy of dependable and secure computing." In: *Dependable and Secure Computing, IEEE Transactions on* 1.1, pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2 (cit. on p. 36).

Bellwood, Tom et al. (2002). *The Universal Description, Discovery and Integration (UDDI) specification* (cit. on p. 18).

Burnette, Ed (2009). *Hello, Android: introducing Google's mobile development platform*. Pragmatic Bookshelf (cit. on p. 5).

Chen, Feng, Dongyun Jin, et al. (2009). "Monitoring oriented programming-a project overview." In: *Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS 2009)*, pp. 72–77 (cit. on p. 17).

Chen, Feng and Grigore Roşu (2005). "Java-MOP: A monitoring oriented programming environment for Java." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, pp. 546–550 (cit. on p. 15).

Chen, Liming and Algirdas Avizienis (1978). "N-version programming: A fault-tolerance approach to reliability of software operation." In: *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pp. 3–9 (cit. on p. 34).

Christensen, Erik et al. (2001). *Web services description language (WSDL) 1.1* (cit. on p. 18).

Delgado, N., A.Q. Gates, and S. Roach (2004). "A taxonomy and catalog of runtime software-fault monitoring tools." In: *Software Engineering, IEEE Transactions on* 30.12, pp. 859–872. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.91 (cit. on pp. 12, 14, 15).

Falcone, Yliès and Sebastian Currea (2012). "Weave droid: aspect-oriented programming on Android devices: fully embedded or in the cloud." In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, pp. 350–353 (cit. on p. 16).

Falcone, Yliès, Sebastian Currea, and Mohamad Jaber (2013). "Runtime Verification and Enforcement for Android Applications with RV-Droid." In: *Runtime Verification*. Springer, pp. 88–95 (cit. on p. 16).

Fowler, Kim (2009). *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Newnes (cit. on p. 47).

Gandhewar, Nisarg and Rahila Sheikh (2010). "Google Android: An emerging software platform for mobile devices." In: *International Journal on Computer Science and Engineering* 1.1, pp. 12–17 (cit. on p. 1).

Hall, Sharon P and Eric Anderson (2009). "Operating systems for mobile computing." In: *Journal of Computing Sciences in Colleges* 25.2, pp. 64–71 (cit. on p. 5).

Jin, Li-jie, Vijay Machiraju, and Akhil Sahai (2002). "Analysis on service level agreement of web services." In: *HP June* (cit. on p. 20).

Keller, Alexander and Heiko Ludwig (2003). "The WSLA framework: Specifying and monitoring service level agreements for web services." In: *Journal of Network and Systems Management* 11.1, pp. 57–81 (cit. on p. 21).

Kumar Maji, A. et al. (2010). "Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian." In: *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pp. 249–258. DOI: 10.1109/ISSRE.2010.45 (cit. on p. 2).

Kwon, Young-Woo and E. Tilevich (2012). "Energy-Efficient and Fault-Tolerant Distributed Mobile Execution." In: *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pp. 586–595. DOI: 10.1109/ICDCS.2012.75 (cit. on p. 24).

La, Hyun Jung and Soo Dong Kim (2009). "A service-based approach to developing Android Mobile Internet Device (MID) applications." In: *Service-Oriented Computing and Applications (SOCA), 2009 IEEE Interna-*

tional Conference on, pp. 1–7. DOI: 10.1109/SOCA.2009.5410278 (cit. on p. 7).

Laprie, J.-C. et al. (July 1990). "Definition and analysis of hardware- and software-fault-tolerant architectures." In: *Computer* 23.7, pp. 39–51. ISSN: 0018-9162. DOI: 10.1109/2.56851 (cit. on p. 37).

Lyu, Michael R. (2007). "Software Reliability Engineering: A Roadmap." In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, pp. 153–170. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.24. URL: http://dx.doi.org/10.1109/FOSE.2007.24 (cit. on p. 29).

Maia, C., L. Nogueira, and L. M. Pinho (2010). "Evaluating android os for embedded real-time systems. In Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications." In: (cit. on p. 2).

Nimodia, C. and Deshmukh (2012). "Android Operating System." In: *Software Engineering, ISSN*, pp. 2229–4007 (cit. on p. 2).

Papazoglou, M.P. (2003). "Service-oriented computing: concepts, characteristics and directions." In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pp. 3–12. DOI: 10.1109/WISE.2003.1254461 (cit. on p. 17).

Powers, R.A. (1995). "Batteries for low power electronics." In: *Proceedings of the IEEE* 83.4, pp. 687–693. ISSN: 0018-9219. DOI: 10.1109/5.371974 (cit. on p. 24).

Rossignol, Joe (2013). *Android Activations surpass one billion*. URL: http://www.businessinsider.com/chart-of-the-day-android-activations-hit-1-billion-2013-9 (visited on 12/23/2013) (cit. on p. 7).

Standarization, International Organization for (2001). *Standard 9126: Software Engineering – Product Quality, part 1* (cit. on p. 18).

Su, S.Y.H. and Edgar Ducasse (Mar. 1980). "A Hardware Redundancy Reconfiguration Scheme for Tolerating Multiple Module Failures." In: *Computers, IEEE Transactions on* C-29.3, pp. 254–258. ISSN: 0018-9340. DOI: 10.1109/TC.1980.1675557 (cit. on p. 32).

Terasa, Clemens and Sibylle Schupp (n.d.). "Annotation-guided soft-error injection." In: *Proc. 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES'13), ser. LNI, to appear* (cit. on p. 39).

Torres-Pomales, Wilfredo et al. (2000). "Software fault tolerance: A tutorial." In: *NASA Technical Report, NASA-2000-tm210616* (cit. on p. 34).

Voas, J.M. (June 1998). "Certifying off-the-shelf software components." In: *Computer* 31.6, pp. 53–59. ISSN: 0018-9162. DOI: 10.1109/2.683008 (cit. on p. 38).

Wang, Le and J. Manner (Dec. 2010). "Energy Consumption Analysis of WLAN, 2G and 3G interfaces." In: *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pp. 300–307. DOI: 10.1109/GreenCom-CPSCom.2010.81 (cit. on p. 26).

Wikipedia (2013). *Runtime verification — Wikipedia, The Free Encyclopedia*. URL: http://en.wikipedia.org/w/index.php?title=Runtime_verification&oldid=549424499 (visited on 12/08/2013) (cit. on p. 12).