Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Bc. Kristián Kacz

Context Aware Android Application Trace Analysis

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Tomáš Pop
Study programme:	Computer Science
Specialization:	Software Systems

Prague 2013

I would like to thank my supervisor Tomáš Pop for his time and valuable suggestions. I also thank to my family for their support and patience during my studies and to my friends for supporting me with test devices.

Název práce: Context Aware Android Application Trace Analysis

Autor: Bc. Kristián Kacz

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Tomáš Pop, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Diplomová práce prozkoumává podporu context-aware aplikací v současných mobilních operačních systémech a vyšetřuje možnosti debugování mobilních aplikací. Práce poukazuje na problémy vyskytující se při debugování context-aware aplikací. Hlavním cílem práce je nejdříve navrhnout debugovací metodu, která bere do úvahy kontextové informace, a pak tuto metodu naimplementovat. Součástí práce je i příklad použití z reálného světa, která demonstruje navrhnutou metodu.

Klíčová slova: Android, Vývoj mobilních aplikací, Debugování mobilních aplikací, Context-awareness

Title: Context Aware Android Application Trace Analysis

Author: Bc. Kristián Kacz

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Pop, Department of Distributed and Dependable Systems

Abstract: The thesis examines how current mobile operating systems support context-aware applications and investigates the methods of mobile application debugging. The thesis points out what kind of problems need to be solved during debugging of context-aware applications. The primary goal of the thesis is to propose a debugging method which takes context information into account and to implement this method. The thesis contains a real world use case to demosnstrate the proposed method.

Keywords: Android, Mobile application development, Mobile application debugging, Context-awareness

Contents

1. Introduction

Mobile devices are one of the fastest developing segments of the computer industry nowadays. Since the introduction of the first iPhone in 2007 and the release of the Android operating system in 2008, the market of smartphones has been steadily expanding. To illustrate this the number of mobile devices sold over time and the market share of mobile operating systems are shown in Table 1.1.

The continuing spread of smartphones and tablets has been a big step towards the world of ubiquitous computing [1]. The human-computer interface has moved from the classical desktop model towards the interaction with 'things that think'. Software for these devices has become an everyday part of almost everyone‘s life. As the number of mobile devices has grown, it has created a huge market for mobile applications. The official application distribution platforms of major mobile operating systems are always reporting an even more rapidly growing software base and download numbers, which are shown in Figure 1.1.

Besides of the large user base, mobile applications have brought into focus one more exciting perspective: context-awareness. The software used on mobile devices can behave differently in various situations, for example: a public transport journey planner application can suggest a source station based on the phone's current geographical location. If no such information is available due to service unavailability, suggestions can be made based on previous user behaviour - recent searches, searches made at a similar time of day, etc. Most of this information is available through the device's sensors such as: location, proximity, orientation,

Year	Total devices sold (millions)	Symbian	Android	RIM	iOS	Microsoft	Others
2008	139.2	52.4%	N/A	16.6%	8.2%	11.8%	10.5%
2009	172.3	46.9%	3.9%	19.9%	14.4%	8.7%	6.1%
2010	296.6	37.6%	22.7%	16.0%	15.7%	4.2%	3.8%
2011	494.5	16.5%	49.2%	10.3%	18.8%	1.8%	3.3%
2012	722.4	3.3%	68.8%	4.5%	18.8%	2.5%	2.1%
1Q13	210.0	0.6%	74.4%	3.0%	18.2%	2.9%	1.0%

Table 1.1: Number of mobile devices sold over time and the market share of mobile operating systems (sources: [2],[3],[4],[5],[6])

etc. Other pieces of context can describe the availability of services, such as network connectivity, or the telephony signal. In comparison to desktop systems the context in which the software is executed significantly less stable.

Context-aware applications drive their control flow based on this type of information. This means that apps can contain bugs which are dependent on a specific context. Furthermore these bugs can be also hard to reproduce, since we need to know the context in which or by which they were caused.

In this thesis, we will investigate the potential of context dependent debugging: a promising method allowing analysis and debugging of mobile device software to find bugs in execution use cases reached only under a particular context.

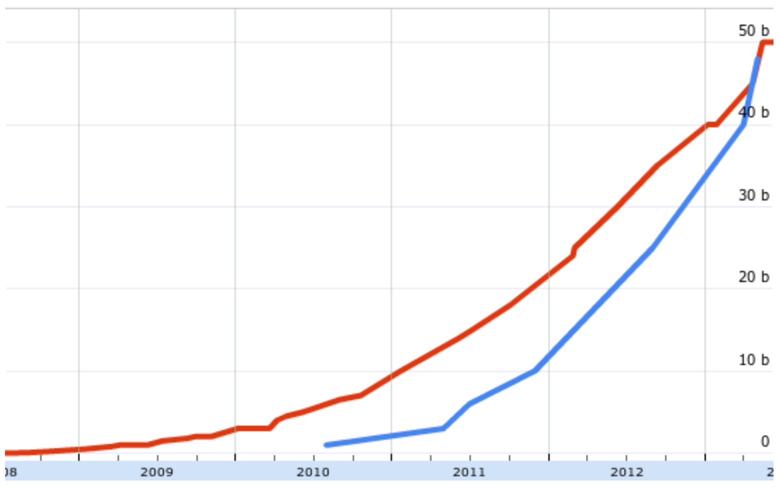


Figure 1.1: Number of downloaded applications in Google Play (blue) and Apple App Store (red) (source:[7], [8])

1.1 Structure of the thesis

Chapter 2 provides a background to context-awareness and examines how mobile platforms deal with context. Chapter 3 examines the mobile operating systems' support for debugging applications. Chapter 4 defines the goals of this thesis and then Chapter 5 discusses the design decisions I have made in completing this thesis. Chapter 6 describes in depth the important parts of the implementation. Chapter 7 demonstrates the the project in use. Chapter 8 draws some conclusions and outlines possible future work.

2. Background

In this chapter we describe the term context-awareness and we show how current mobile operating systems deal with context.

2.1 Context-awareness

Computer software by itself is a set of machine-readable instructions and operates algorithmically based on mathematical rules. It can process input, return output and interact with other entities. As computing has weaved into our society by the spread of smart devices, our expectations placed on the behaviour of software running on these gadgets have changed. An example of such expectation is that devices should be aware of the environment surrounding them and adapt to it [9], in other words to be context-aware. By 'adaptation' we mean selecting, processing and presenting relevant information according to the given situation.

Context for a mobile application is the set of information describing its environment. Dey and Abowd in [10] defined it as

"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

This definition, however, restricts context to describing only relevant entities that are in interaction with the application. But non-interactive applications can also be dependent on context. A more universal definition is formalized by Wei and Chan in [11]:

"Context is application specific, and context of an application is any external information which can be utilized to adapt the data, behavior or structure of this application."

This definition points out that contexts are application specific - information relevant to one application does not have to be relevant to another. Another important thing is that context is external to an application. Information coming from within the application can be dependent on parts of the context, but the information itself is not a part of the context. According to Wei and Chan, context can be divided into three categories, these are physical context, computing context and user context.

Physical context describes physical properties of the device's environment. They can be obtained by specially designed physical sensors such as accelerometer, gyroscope, etc. Most modern mobile devices are equipped with many sensors. Measured data is usually available to the software through the API of the operating system.

Computing context consists of runtime information about the device hardware. Operating systems usually provide primitives to monitor CPU, memory usage and network conditions; however, the coverage of all these data varies a great deal between different systems.

User context describes the actual state of the user of the device from a higher-level perspective, such as location (at work, at home, etc.), activity (e.g. working, driving or travelling). These are not directly measurable; instead they are defined as a composition of previously described contexts.

2.2 Android

Android is an open source operating system originally released for touchscreen mobile devices, now available on a wide range of electronics. It was released in September 2008 by Open Handset Alliance, a consortium of 84 hardware, software and telecommunication companies with the aim to develop open standards for mobile devices. The source code was released under the Apache public licence and is maintained by the Android Open Source Project community.

Android consists of a Linux based kernel, middleware, libraries and APIs. The architecture of the system is shown in Figure 2.1. Applications written in Java language are compiled into bytecode which is then translated into *.dex* (Dalvik executable) files. These run in the Dalvik virtual machine, Google's implementation of Java runtime. Applications can also call native code written in C/C++, which is compiled with the Android NDK. All these files are packaged together into an *.apk* Android package file. The package also contains a *manifest* *file* with a description of the application for the system. It defines, among other things, the permissions needed by the application, entry points to the application, etc.

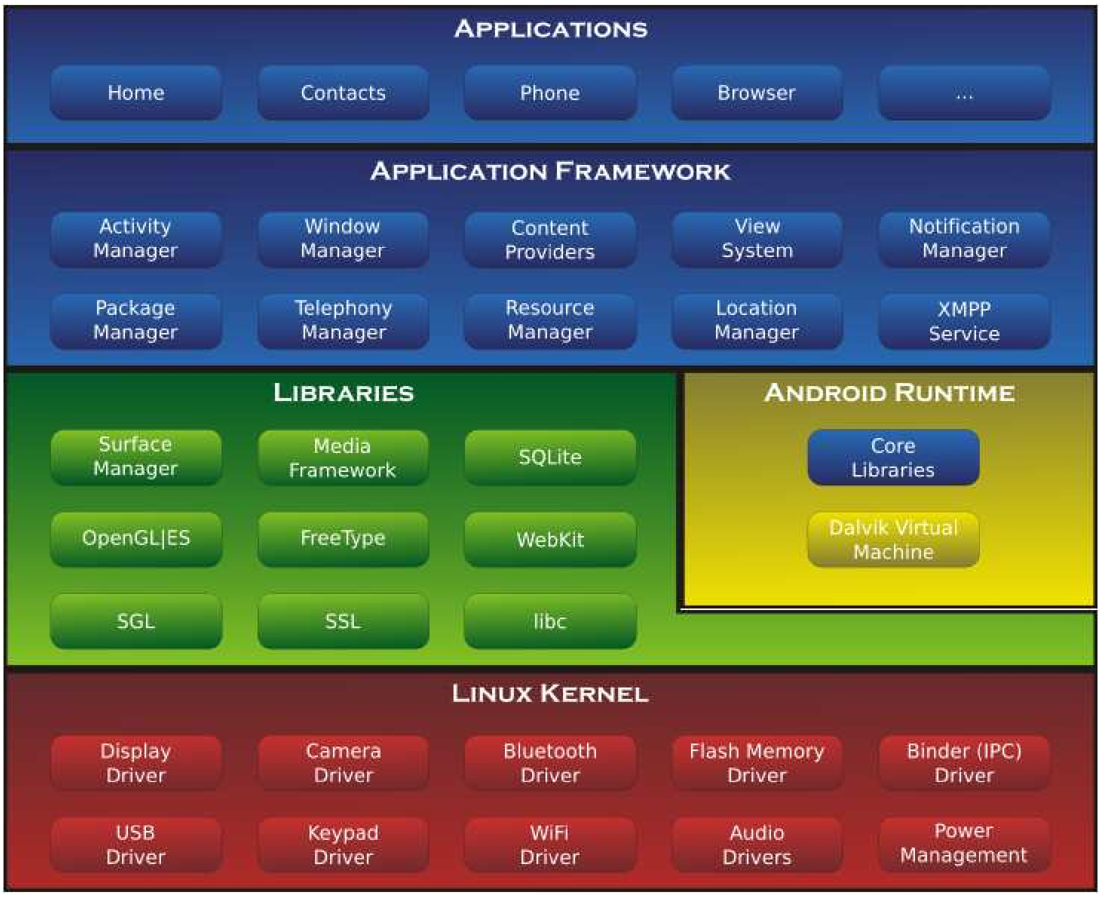


Figure 2.1: Architecture of the Android operating system (source:[32])

Security sandboxing of running applications applied by the Android system implements the *principle of least privilege*. Each application lives in a separate virtual machine, running in its own Linux process. By default every application has a unique Linux user ID and has permission to access only the files assigned to this ID.

The building blocks of Android applications are *components*. Each of them is a different entry point to the application. The four types of components are activities, services, content providers and broadcast receivers.

An activity represents a single screen of the user interface. In applications having more screens, activities call one another. They can also be called from other applications if they are registered as providing some feature. For example, a directory listing activity of a file browser application can be used in other applications requiring file selection dialogs. These entry points are described in the manifest file of the application.

Services are components without a user interface, running in the background. They are usually performing long-running computations. Other components can either start a service or bind to it. Started services are running in the background performing a single operation. When the operation is finished the service should stop itself. Bound services are working on a client-server basis. Other application components can communicate with them by sending requests and getting back results. They can even run in separate processes and communicate by inter-process communication. A bound service is destroyed when all the components are unbound from it.

Content providers are the application's main access points to structured data. They provide an interface for storing data on persistent locations such as an SQLite database, a file in the filesystem or on the web. If the content provider allows, other applications can also access and modify stored data.

Broadcast receivers are components for accessing system-wide announcements, for example, context changes. Announcements have the form of an `Intent` object. Broadcasted Intents are sent to all receivers registered for the given type of intent.

2.3 Context-awareness in Android

The Android system has a rich support for context-awareness. Part of the application context is represented in the system by the class `android.content.Context`, which is the interface to global information about the environment. It has methods to access system services through which we can query the device's hardware for sensor data and external service availability. Activities and Services are inherited from this class. Other classes need a reference to be passed to access the context.

Most of the information available through this interface is part of the physical context. `SensorManager` obtained from the Context class is the main access point to the Android sensor framework. The available sensor list depends on the version of the system. According to [12] on Android 4.0 the following sensors are supported: accelerometer, ambient thermometer, gravimeter, gyroscope, linear accelerometer, magnetometer, orientation sensor, barometer, proximity meter, relative humidity meter, rotation sensor and thermometer. Most smartphones are equipped with the majority of these sensors; however, some sensors are present only in very specialized devices.

Among the physical context some parts of the computing context is also available through the `Context` interface.

`ConnectivityManager` answers queries about the state of data network connectivity. The data network can work over various physical network layers such as Wi-Fi, GPRS, or Bluetooth. For querying about these we can use `WifiManager` and `TelephonyManager`.

`DisplayManager` and `AudioManager` provide access to a query for the audio-visual state of the device.

`LocationManager` provides access to the location services of the system. Location information can be derived from more than one sources. GPS, Wi-Fi and cell data can each provide information on location. Each of them, however, has different accuracy, speed and battery usage.

Other parts of the computing context can be obtained by querying the settings API of Android or accessing the underlying Linux operating system.

`Setting.System` class provides access to various system preferences such as ringer volume, screen brightness and so on. Similarly `Settings.Secure` and `Settings.Global` contain read-only and system wide preferences.

Computing context information which are not available through Android APIs can be obtained from the Linux interface; for instance, `/proc` contains information about the running processes.

2.3.1 User context on Android

User context itself is not supported in Android but there are frameworks and libraries that add this feature to the system.

AWARE

AWARE is a framework for Android systems dedicated to instrument, infer, log and share mobile context information [13]. It is available as an Android library to empower rich context-aware applications for users. Its main building blocks are *Context Sensors* and *Context Plugins*. Context Sensors collect physical and computing context data from the hardware and software sensors. They can also reuse data from other Context Sensors to provide higher-level context data like user context. Context Plugins are used for presenting data to the user. The architecture of the framework is designed keeping extensibility in mind; therefore all these components are implementing an add-on interface. The collected context information is accessible for applications through *Context Broadcasts*, *Context Providers* and *Context Observers*. Context Broadcasts are notifications used internally by the framework to update other sensors and plugins. They can also be captured by other applications. This is the 'lightest' way to receive data from AWARE. Context Providers are extensions to Android's `ContentProvider`, and therefore can be used to store context data on the device. Stored data can be accessed by an application with Android's `Cursor`. This access method is passive, since applications need to actively query for information. Conversely, Content Observers, which are extensions to `ContentObservers`, provide an active and event-driven access method to context data. They can be registered to Providers in order to get automatically notified when new data is available.

Funf framework

The Funf Open Sensing Framework is an open source tool for collecting and processing context information [14]. It is developed by the Behavio team from MIT which is now part of Google.

The framework consists of a set of Probes which are objects collecting data from the device's sensors and GPS, but also data about call logs, application usage and browsing information. The framework can be extended by adding third party probes. Logged data can be exported to the SD card, or the framework can be connected with Dropbox to automate the upload of log files. The framework can also be installed as a standalone application running in the background. This application also allows the user to change the enabled probes during runtime.

Morphone.OS

Morphone.OS is a context-aware Android-based mobile operating system [15]. It is based on the idea of a *self-aware computing system*, which means systems that can observe their runtime behaviour, learn and take actions to meet desired goals. These three steps (observing, decision making and acting) are realized in the core of the system [16]. Observation is based on the Heartbeats API. Applications at key moments issue a heartbeat by a simple function call. The performance of the application is therefore its heart rate. Applications can also register their desired performance: this will be the goal to achieve. Acting is done by services, which can perform changes in applications or in the whole system. Such services are, for example, the core allocator, frequency manager or adaptive scheduler. Decision making is done by the service coordinator. It collects data about the application and system performance and starts or stops acting services in order to get closer to the desired goal.

The system has services included to monitor the context. All three defined context categories are covered. Captured physical and computing context data include sensor data, location, CPU usage, process status, etc. Covered user context data include, for example, frequent user actions and common situations. Morphone applications can also access these pieces of information and use them for decision making.

Gimbal

Gimbal is a context-aware platform for Android and iOS [17]. Its main feature is *Geofence* which allows software developers to create digital boundaries around physical spaces. This makes it easier to develop rich geographically aware applications. When the user enters an area specified by the developer, the application can request news about the places nearby and notify the user, or just simply remember their entrance to refine their personal preferences. This requires Gimbal's next important feature, namely, Interest Sensing. Based on finely tuned heuristics, Interest Sensing is able to create a user profile that helps the application to best match a customer's specific preferences and display information when they are most interested.

2.4 Context-awareness on Windows Phone

Windows Phone 8 is the current version of Microsoft's mobile operating system. Its architecture is based on the Windows NT kernel, the same as the desktop operating system Windows 8. This makes it possible to easily port applications between the desktop and the mobile platform.

Context-awareness is supported on Windows Phone by system APIs. Location API gives applications access to the geographical locations of the device. Apps can make one-time location requests with specified accuracy and maximal waiting time or can subscribe to receive location updates after a given interval or after the device has moved a given distance from the last known location. Sensor API provides access to physical sensors of the device. Supported sensors in WP8 are accelerometer, gyrometer, compass, inclinometer and orientation sensor. Applications can access raw data from the first two of them. Data from the other sensors are combined together by sensor fusion applying mathematical calculations to correct sensor limitations and reduce noise (see Figure 2.2).

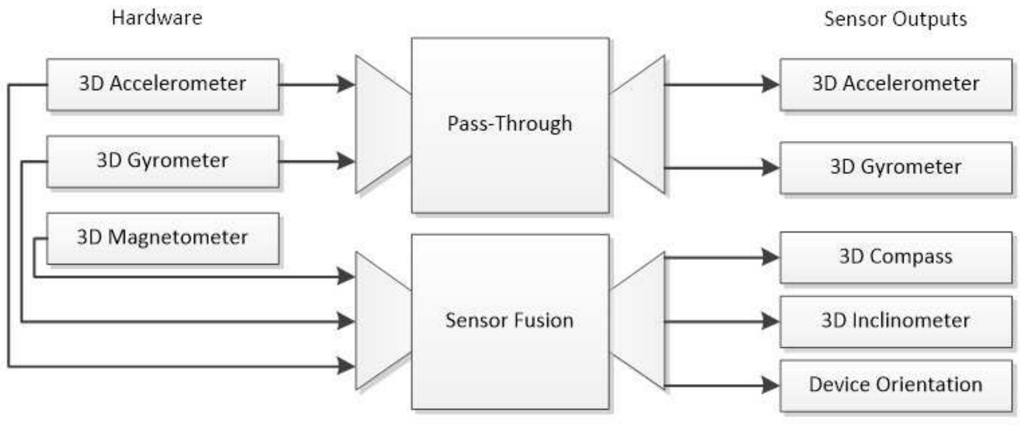


Figure 2.2: Windows Phone 8 sensor fusion system (source:[33])

Computing context is only partially covered in the Windows Phone platform. Stats about memory usage and power source are available through the `DeviceStatus` class from the `Microsoft.Phone.Info` namespace. Network availability can be queried via the `NetworkInformation` class from the

`Windows.Networking.Connectivity` namespace. Battery information can be accessed with the `Battery` class from `Windows.Phone.Devices.Power` namespace. CPU usage statistics, however, can not be acquired through the SDK.

Microsoft aims to increase the support of context-awareness in their mobile operating systems. They introduced the idea of *Contextual Data Units* (CDUs) which are an abstract data type representing a meaningful unit of context for applications instead of raw sensor data [18]. An example CDU is *user motion state* with possible values as sitting, standing, walking, running, etc. According to the definition this can be viewed as a part of user context. Their proposed Context Dataflow Operating System (CondOS) incorporates CDU generation in the kernel of the system. This makes it possible not only for applications but also for system services to respond on context changes. Thanks to this, memory management, scheduling, energy management, security can be context-aware too.

- Context-aware memory management lets the system to choose which applications to preload. For example, if the system determines user's motion state as 'driving', it should preload navigation and maps applications. Instead, if the motion state is 'running', preloaded apps should be a music player and workout app. Similarly, memory eviction can also use context data. The decision as to which background apps to unload when there is not enough memory for starting applications is usually based on a LRU algorithm. However, during 'driving' state games will unlikely be used, so they should be probably unloaded.

- Security management can also benefit from using context information. If the device notices that the user's current location is 'home', security manager can relax the security level by switching off password protection and enabling content sharing between devices. On the other hand, biometric context can help to recognise whether the device is held by its owner or is used by someone else in which case applications containing sensitive personal data, such as a contact list, or call and message log, can be disabled.

- Modern mobile operating systems schedulers usually use a priority-based preemptive scheduler. High priority is often assigned to the foreground process while other processes are starved. Context-awareness can improve this method by letting apps specify the priority level for each context they want to run in. This makes possible for background processes to get higher priority in situations they are associated with.

- Context-aware energy management can use location information to predict time-to-recharge. This helps to save the battery by disabling energy consuming background tasks when the user is out and about.

CondOS by default contains a set of CDUs. The applications, however, may need additional pieces of context. For this reason developers can implement their own CDU Generators and install them into the kernel of the system. Generators are directed acyclic graphs of processing components. Components are sandboxed pieces of code and communicate only through producer-consumer interfaces. The first component takes raw sensor data as input and the last produces CDUs as output. The design of generators is shown in Figure 2.3. Applications can make one-time CDU requests or subscribe to some CDUs.

2.5 Context-awareness on iPhone

iPhone is a series of smartphones developed by Apple. The innovations introduced in the first generation, such as the big touchscreen with multi-touch, or the set of sensors have become the de facto standard on smartphones. Its current version, iPhone 5 was released in September 2012 and is running the iOS 6 operating system.

Applications on iOS are written in Objective-C.

iPhone applications can obtain location information from the `CLLocationManager` class from the Core Location framework. It can also report changes in the device's heading. Data from other sensors are available through the Core Motion framework. `CMMotionManager` provides access to raw accelerometer, gyroscope and magnetometer data and computed device motion information, derived from the three mentioned sources. `UIDevice` class provides some parts of the physical context, like proximity sensor data and device orientation. It gives access also to parts of computing context, namely battery status and information about the

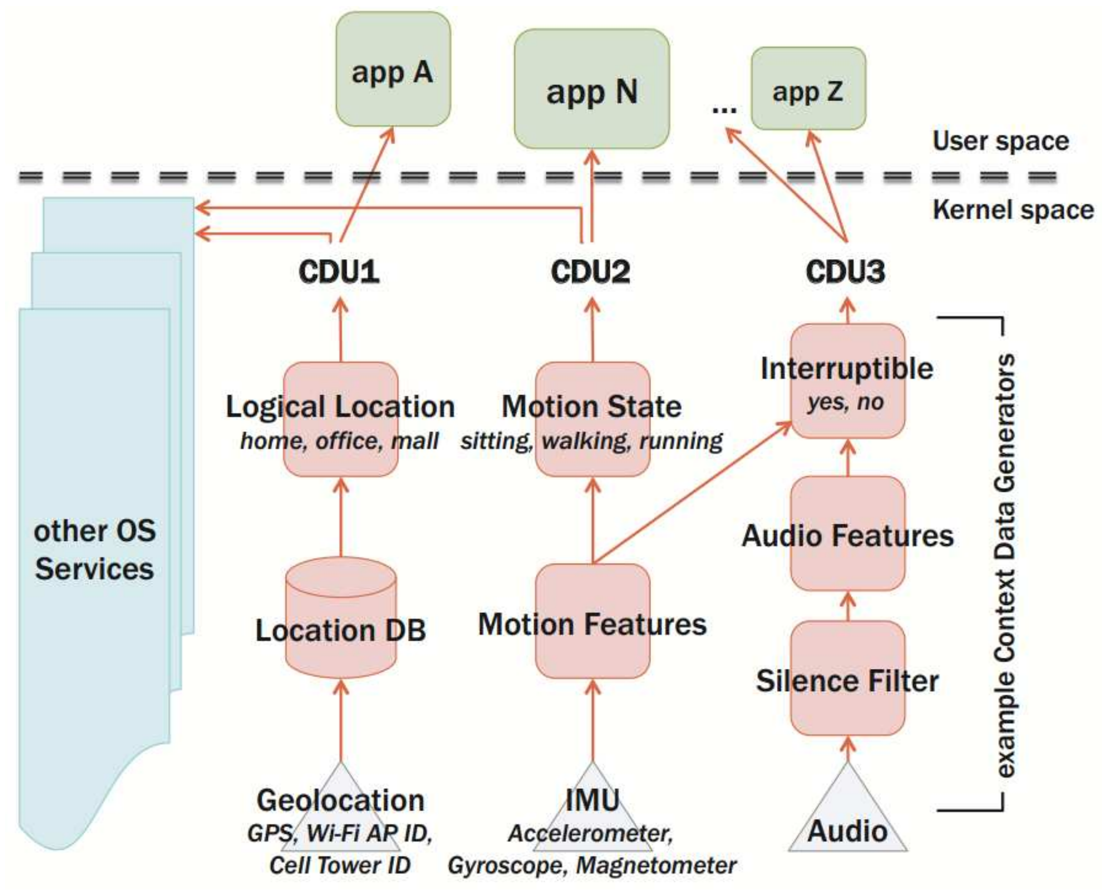


Figure 2.3: CondOS dataflow example (source:[18])

operating system. Other parts of computing context are not covered by public APIs; however, there are low-level functions defined in the Mach API, which can return such information [24]. `host_processor_info()` returns the number of CPUs and their utilization, and `task_info()` returns information about memory usage of current task.

Higher level context information is not supported directly in the operating system; however, third-party libraries exist to provide this feature. Gimbal is an example mentioned earlier.

3. Debugging and tracing

We will now show what tools can help developers to debug their applications on various platforms.

3.1 Android

The Android SDK contains a set of tools for debugging [19].

Android Debug Bridge (adb) is a command line tool responsible for communication between the device and the development environment. It consists of three parts:

- a server process running in the background on the development machine. It sets up a connection to the devices and passes commands to them.
- clients running on the development machine sending requests through the adb server to the devices. Clients can be created by the developer by calling adb from the shell or by tools like the ADT plugin of the IDE or the DDMS.
- a daemon running on the mobile device, accepting a connection from the adb server and responding to the requests.

Figure 3.1 shows how these parts work together.

Dalvik Debug Monitor Server (DDMS) is a debugging tool providing features such as port-forwarding, screen capture, thread and process information monitoring, incoming call, SMS and location data spoofing. It uses adb to connect to the debugging port of the VM running on the device. When connected, a JDWP compatible debugger can be attached to debug the application on the device.

Android Developer Tools (ADT) is a plugin for the Eclipse IDE, integrating a set of tools into the development environment, such as DDMS and Logcat. Applications can be easily compiled, packaged and deployed to a device or emulator by clicking Eclipse's run button. The IDE's debugger can be attached to running applications to stop them on breakpoints.

LogCat is an adb command to access the system debug output. The Android logging system provides a mechanism to write messages to the system log using

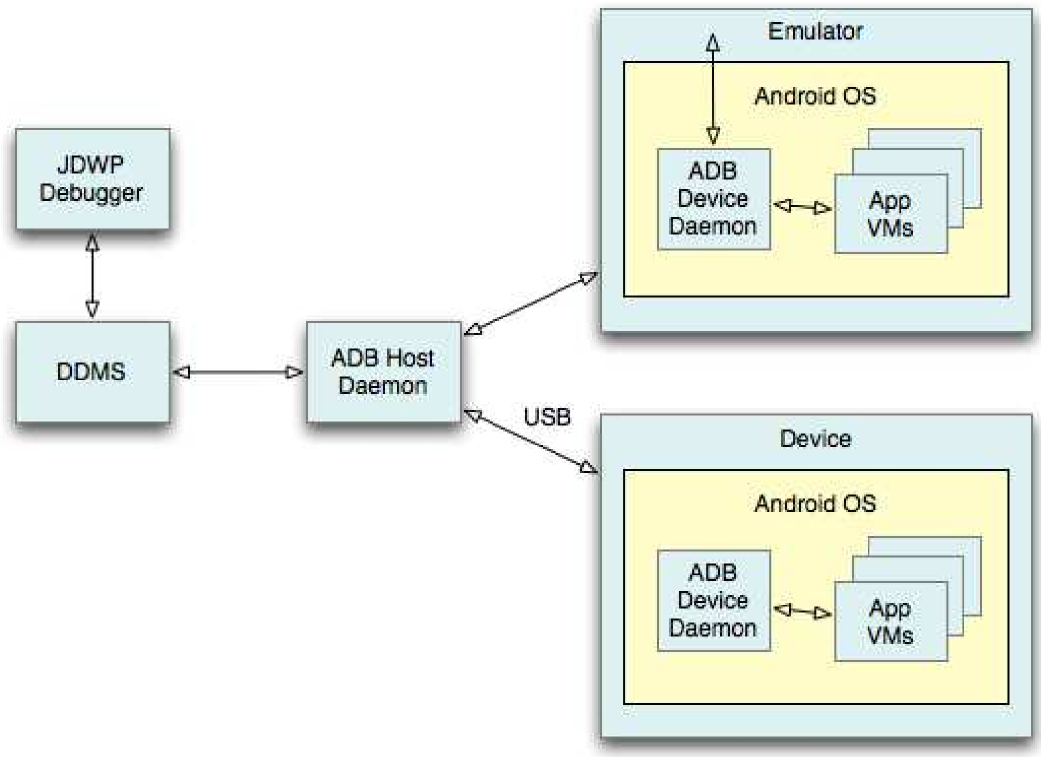


Figure 3.1: Cooperation of Android SDK debugging tools (source:[19])

the `Log` class from the `android.util` package. Each log message has *priority level* and *tag* defined. Priority can have one of the following values (in ascending order): verbose, debug, info, warning, error, fatal. Tag describes the source of the message. Messages are written into different circular buffers. Three main buffers are `radio` for telephony related messages, `events` for event-related ones and `main` for all messages. When LogCat is connected to the device, it can read messages in real time. Messages can be also filtered by tag and minimum priority level.

The drawback of the above-described methods is that the development machine needs to be connected to the device to collect debug information. However, offline data collection is also supported on Android. `Debug` class from the `android.os` package contains the function `startMethodTracing()` which triggers the virtual machine to start logging every entry and exit from every function. Since it is implemented on the VM level, the application code does not need to be instrumented. However, native code running outside of the VM can not be logged directly. The trace will only contain calls of the Java function with a native keyword.

Calling the `stopMethodTracing()` method on `Debug` class will stop the tracing and save the results.

The output of the tracing is a binary file containing records of every function call during the tracing period. The file follows the format described on address [20]. It is built up from two parts: the key part and the data part. The key part contains three sections. The first section describes the version of the trace file and the clock mechanism used in trace records. The next section contains information about the threads of the application. Finally, the third section describes the methods by its containing class, name and signature.

The data part of the trace file contains a header, describing its version and start time of tracing as a value coming from the `gettimeofday()` function. After the header a list of records follows. Each record contains a thread ID, a method ID, a method action and a timestamp. Method action can be one of *entry*, *exit* and *exception thrown*. Timestamp represents delta time in microseconds since the start of tracing.

The trace file can be processed by many ways. Android SDK contains two applications for this purpose.

`dmtracedump` is a tool for generating tree diagrams from the trace file as shown in Figure 3.2. Each node represents a call, with method name, inclusive/exclusive times and number of calls displayed.

The other application for processing the trace log is `Traceview` shown in Figure 3.3. It has two main panels. The timeline panel shows the execution of each thread in the app's process. Each method has a different colour. If a method is selected, then lines appear under the row showing the extent of all the calls of the selected method. This panel can be useful during the debugging of an application. The profile panel shows a summary of all the time spent in each method. This can be useful during profiling of the application.

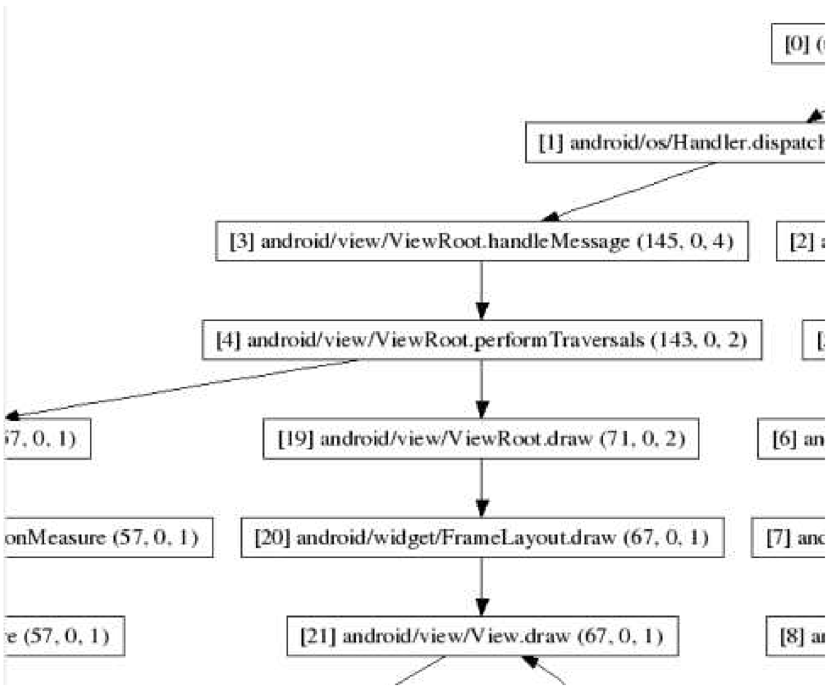
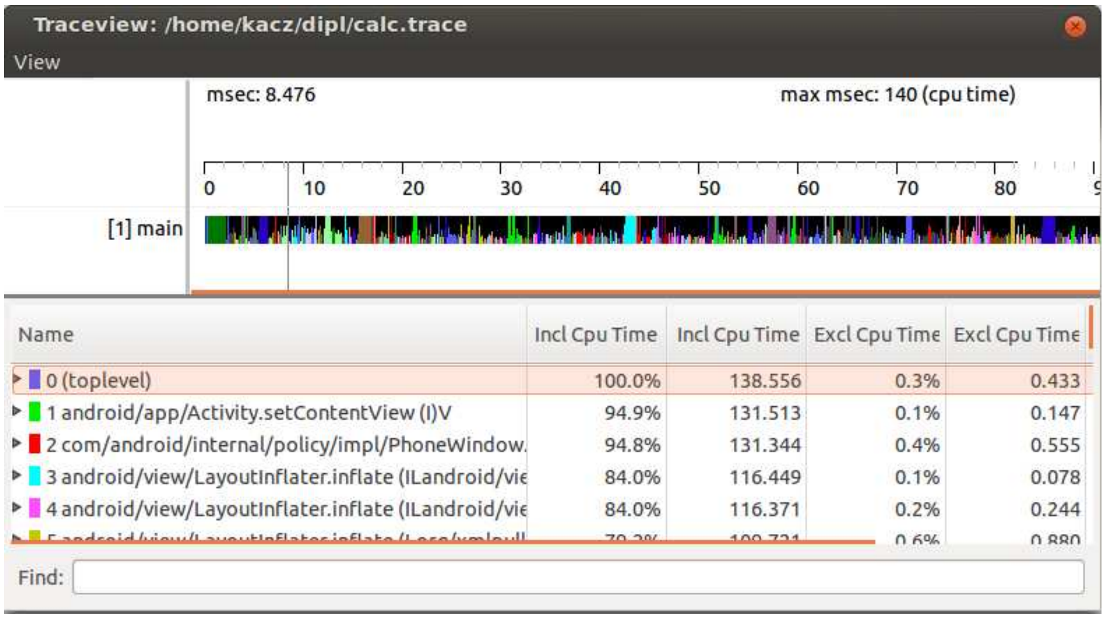

[0] (
[1] android/os/Handler.dispatch
[3] android/view/ViewRoot.handleMessage (145, 0, 4)
[2]
[4] android/view/ViewRoot.performTraversals (143, 0, 2)
[
57, 0, 1)
[19] android/view/ViewRoot.draw (71, 0, 2)
[6] an
onMeasure (57, 0, 1)
[20] android/widget/FrameLayout.draw (67, 0, 1)
[7] anc
e (57, 0, 1)
[21] android/view/View.draw (67, 0, 1)
[8] a

Figure 3.2: Screenshot of dmtracedump (source:[20])



Name	Incl Cpu Time	Incl Cpu Time	Excl Cpu Time	Excl Cpu Time
0 (toplevel)	100.0%	138.556	0.3%	0.433
1 android/app/Activity.setContentView (I)V	94.9%	131.513	0.1%	0.147
2 com/android/internal/policy/impl/PhoneWindow.	94.8%	131.344	0.4%	0.555
3 android/view/LayoutInflater.inflate (ILandroid/vie	84.0%	116.449	0.1%	0.078
4 android/view/LayoutInflater.inflate (ILandroid/vie	84.0%	116.371	0.2%	0.244
5 android/view/LayoutInflater.inflate (Lorg/xmlpull	70.3%	100.721	0.6%	0.880

Figure 3.3: Screenshot of Traceview

3.2 Windows Phone

Similar to Android, the Windows Phone SDK has debugging tools integrated into the development environment.

`Debug` and `Trace` classes from the `System.Diagnostics` namespace provide methods to log messages in debug/release modes, respectively [21]. Messages are received by objects from the `Listeners` collection, implementing the `TraceListener` interface. These objects emit the messages to the Win32 `OutputDebugString` function and to the `Debugger.Log` method and therefore will appear in the connected debuggers console.

The drawback of this solution is that application code needs to be manually instrumented to log function calls and that the device needs to be connected to the development machine running Visual Studio.

To overcome the need for manual instrumentation, IntelliTrace could be used [22]; however, it's current version does not support tracing Windows Phone apps. It is a tool for recording code execution, integrated into Visual Studio. With a recorded trace, debugger can be used without executing the application again.

The second drawback can also be resolved by using commercial software such as DevTracer [23], which can send the log messages through a web service to its own trace monitor for further analysis. This eliminates the need of being connected to the development machine.

`System.Diagnostics` contains also the `StackTrace` class which represents information about the current stack trace of the calling thread. However, use of this class for collecting trace information would also require manual instrumentation.

3.3 iOS

`NSLog` function from `FoundationKit` provides basically the same functionality as the classes mentioned earlier: code can be instrumented by it to write log messages to the console. `NSLogToFile` immediately forwards these logs to a file. Using this function, however, again needs instrumentation of the application code.

4. Problem statement and goals of the thesis

As we have already shown, mobile operating systems support applications to use contextual data. Since the higher level context, such as parts of the user context, is usually not covered by default in these systems, applications need to process the physical and computing context themselves in order to generate the higher level information they need. However, if this generation of higher information contains a bug, it can lead the application to behave as it should in a different context. On the other hand, if a bug is hidden in some other piece of code which only runs in a specific context, it can be difficult to reproduce the error. This makes it hard to investigate the erroneous behaviour of buggy context-aware applications.

We have already examined how mobile platforms currently support debugging. Most of the techniques require the device to be connected to the development machine. This can be a narrow constraint during the development of context-aware applications since some erroneous behaviour can appear in situations when the development machine is not available. Saving a trace of the code execution can solve this problem; however, it usually needs some instrumentation of code to log every function call. None of the debugging tools, however, take into consideration context data.

The thesis has two main research goals:

1. To propose a debugging method that takes context data into account. We will design and implement a library for mobile applications to log context data and trace code execution during the execution of the software. The library should be easily linked to existing Android projects, without heavy instrumentation of the application's code. It should cover the most frequently used context information and also should be easily extendible to log user defined context.

2. To implement a debugging tool to process logged data. The tool should be able to visually combine the logged execution trace and the context data. It should also give hints about which contexts could have caused the erroneous behaviour.

5. Analysis

In this chapter we will discuss design decisions made during the preparation of the project.

5.1 Platform

The first and also the most important decision during the development was the choice of platform. Three platforms were considered: Android, Windows Phone and iOS. All of them have similar support for context-aware applications: they give access to sensor data, and the application needs to process them in order to get context information.

From the debugging point of view, Android provides the best features. It makes it possible to log the execution trace to a file on the SD card without the need of permanently being connected to the development machine. Saving such a trace also does not need heavy instrumentation in the code of the application, since it's logging can be started with a single function call.

Another big advantage of Android is that the whole system and the SDK are released under an open source license. This makes it possible to integrate our solution to an existing debugging tool.

After summarizing these facts we came to the decision to select the Android platform for our project.

5.2 Collecting trace information

As mentioned earlier, Android has built-in support for logging the execution trace of an application. It can be started simply by calling the function `startMethodTracing()` of `android.os.Debug` class and stopped by calling `stopMethodTracing()` function of the same class.

While tracing is enabled, the instance of the virtual machine hosting the application process logs every function call and function exit of the hosted processes. Therefore, if we want to trace only the application code itself and not to trace the code of our library collecting the context information, it needs to be executed in a separate process running in a different virtual machine instance. The ideal solution for collecting context data would be the use of a background service running in a new process. By default, all components of an application are running in the same process. This can, however, be overridden in the manifest file of the application [25].

5.3 Covered context-data

We wanted the context logging mechanism to be adjustable to the needs of a specific application. Our aim, therefore, was to cover a comprehensive range of loggable context information but at the same time let the developer choose what exact pieces of context he wants to log. We split up all loggable context information into categories and implemented each category as a separate listener. The categories should contain pieces of context that often change together, such as Wi-Fi connection status and SSID. We also realized that some applications can request special pieces of context that are not covered by us. Therefore we designed an interface that makes it possible for developers to implement their own listeners and use them with our library.

Implementation will be described in depth in the next chapter.

5.4 Output of logging

The output of the logging process will be a pair of log files. The first of these is the output generated by the `startMethodTracing()` function. The second file will contain the logged context information. The structure of the file will be similar to the trace file structure for easing the processing of the two files together. The exact structure will be described in the next chapter.

5.5 Synchronisation of logged data

Since the data in the two files will be processed together, they need to be synchronizable.

The trace file contains the timestamp of the start of the tracing. It's value comes from the `gettimeofday()` function. Every record in the file also contains a timestamp. The source of this time information depends on the version of the system. According to the trace file documentation [20] the current version of the system supports only the global clock and per-thread CPU times will be implemented in future versions. However, this page is outdated. The source code of the proper trace file generating function contains a preprocessor directive enabling per-thread clock usage since the earliest available versions.

- Android 1.6 (Donut) - 2.2 (Froyo):
  the choice of the time source is made at compile time of the Dalvik virtual machine with a preprocessor directive `HAVE_POSIX_CLOCKS` (see Code snippet 5.1). If it is defined, the system will use the `clock_gettime()` function with `CLOCK_THREAD_CPUTIME_ID` as the clock id parameter to get per-thread CPU time. Otherwise, if the directive is not defined, `gettimeofday()` will be used as the time source for the records too. The definition of the directive is contained in the architecture-specific `AndroidConfig.h` file. For the ARM architecture the directive is defined in all versions of the system, so devices with these versions use `clock_gettime()` as the time source.

```
static inline u8 getClock()
{
#if defined(HAVE_POSIX_CLOCKS)
  struct timespec tm;
  clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tm);
  //assert(tm.tv_nsec >= 0 && tm.tv_nsec < 1*1000*1000*1000);
  if (!(tm.tv_nsec >= 0 && tm.tv_nsec < 1*1000*1000*1000)) {
    LOGE("bad nsec: %ld\n", tm.tv_nsec);
    dvmAbort();
  }
  return tm.tv_sec * 1000000LL + tm.tv_nsec / 1000;
#else
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec * 1000000LL + tv.tv_usec;
#endif
}
```

Code snippet 5.1: Time source choice in Android 2.2 and earlier (source:[34])

- Android 2.3-2.3.6 (Gingerbread):

  Systems without posix clocks still use `gettimeofday()` as the time source. Systems with `HAVE_POSIX_CLOCKS` defined have a boot time choice of time source. The default option is to use per-thread clocks. However, if Dalvik runtime is started with option `-Xprofile:wallclock`, profiling routines will use `gettimeofday()` too (see Code snippet 5.2). Parameters can be set through the adb command `setprop`. This method requires a restart of the Android runtime and in addition parameters are reset when the device restarts. The same result can be achieved if the line "*dalvik.vm.extra-opts = -Xprofile:wallclock*" is attached to the file `/data/local.prop` [26].

  To use setprop or modify the file root privileges are required.

```
static inline u8 getClock()
{
#if defined(HAVE_POSIX_CLOCKS)
    if (!gDvm.profilerWallClock) {
        struct timespec tm;

        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tm);
        if (!(tm.tv_nsec >= 0 && tm.tv_nsec < 1*1000*1000*1000)) {
            LOGE("bad nsec: %ld\n", tm.tv_nsec);
            dvmAbort();
        }

        return tm.tv_sec * 1000000LL + tm.tv_nsec / 1000;
    } else
#endif
    {
        struct timeval tv;

        gettimeofday(&tv, NULL);
        return tv.tv_sec * 1000000LL + tv.tv_usec;
    }
}
```

Code snippet 5.2: Time source choice in Gingerbread (source:[35])

- Android 3 (Honeycomb):

  since the code of this version was not published and this feature is not well documented, we have no information about the behaviour on this system.

- Android 4.0 (Ice Cream Sandwich) and newer:

  Systems without thread-specific clock support still use `gettimeofday()`. If a system supports per-thread clocks, it can use both sources in logs (see Code snippet 5.3). The choice is made again at the initialization of the system, but the default value is the option is to use dual clock (both wall and per-thread).

```
#if defined(HAVE_POSIX_CLOCKS)
    if (useThreadCpuClock())
    {
        u4 cpuClockDiff = (u4) (getThreadCpuTimeInUsec()
                                    - self->cpuClockBase);
        *ptr++ = (u1) cpuClockDiff;
        *ptr++ = (u1) (cpuClockDiff >> 8);
        *ptr++ = (u1) (cpuClockDiff >> 16);
        *ptr++ = (u1) (cpuClockDiff >> 24);
    }
#endif

    if (useWallClock()) {
        u4 wallClockDiff = (u4) (getWallTimeInUsec()
                                    - state->startWhen);
        *ptr++ = (u1) wallClockDiff;
        *ptr++ = (u1) (wallClockDiff >> 8);
        *ptr++ = (u1) (wallClockDiff >> 16);
        *ptr++ = (u1) (wallClockDiff >> 24);
    }
```

Code snippet 5.3: Time source choice in Ice Cream Sandwich and newer (source:[36])

Thread-specific clocks are useful when profiling, since they give precise information on how much CPU-time was needed to run a given piece of code. We need, however, timestamps for synchronisation of logged trace and context information. For that purpose the wall clock time is more suitable. Therefore our library will need global clock enabled on Gingerbread and older systems and dual clock on Ice Cream Sandwich and newer.

We want to process the trace logs and the context logs together, so we will need to save a timestamp for each record in the context log file too. To be sure that the records from the two files are in sync, we should use the same time source in both files. Therefore our library will use the `gettimeofday()` function for getting timestamps for context logs.

5.6 Timestamp accuracy

Timestamping of trace records is done by the system at the VM level. When tracing is enabled, every method call and method exit triggers a function of Dalvik, which adds a record to the log file. Therefore, from the perspective of the application, the timestamp in the record represents the exact time, when the call or exit actually happened.

During context logging we aim to minimize the delay between the context change event and the actual timestamp taken for the log record.

One way how our library can be notified of context changes is by receiving system broadcasts. These usually contain information about the new context, but no information about the exact time, when the change happened. Other sources of information about context change are SensorEvents. These objects contain a timestamp of the event; however, its value has a different meaning depending on the version of the system. It is either the time since boot or the unix time, both in nanoseconds. This can be used for relative time measurements, but for our use in this case it is not suitable.

Therefore time sampling needs to be done inside our library and ideally with the smallest possible overhead. To achieve this, we will run each listener's code in a separate thread. Threads will be blocked until the proper piece of context is changed. When resumed, the first step of the handler method will be the lookup of current time. This ensures that the timestamp will be as close as possible to the event occurrence.

5.7 Processing logged data

For processing the output of logging we decided to implement a tool based on the existing tools included in the Android SDK. This choice has two advantages: first of all, developers who have already used these tools will be able to easily get familiar with our extended version. Secondly, these tools already implement methods for processing the trace log. This code will be useful for us, and if we follow the structure of the trace file during the design of the context log file, it will help us implement the processing code.

The two tools for trace file processing in the SDK are `Traceview` and `dmtracedump`. Both of them give us a different perspective on the logged trace. dmtracedump visualizes the trace file as a tree graph. Nodes representing functions are connected if they call one another. This type of visualization can be extended with context information: by specifying conditions on some context values, the software could highlight the paths in the tree that were executed while the given conditions were met.

Traceview shows the trace logs in a linear manner over a timeline. Methods can be selected by name, or by clicking on them on the graph. Selected methods get highlighted on the timeline and additional information is shown about them. This type of visualization can be meaningfully extended with context data. Each piece of context should be displayed in a separate row, similar to that of a trace from one thread. Rows should contain visual representation of context data. For each type of logged context data we should choose a proper visualization method:

- floating point data (float and double) - For this kind of data we should use a classical line chart. Records from the log file will represent points with known values. They will be connected with a straight line. Values of the intervening points will be computed with interpolation of values from the two neighbouring records.

- integer data (int and long) - Similarly as for floating point data, logged records will represent points with known values. However, since this type is used to represent discrete data, we cannot connect points with a straight line.

  Thanks to our effort to minimize the delay between the context change event and the taking of the timestamp for the log record, we can assume that our library detects a change in context as soon as possible. Until that time we can presuppose that the value remains unchanged.

  Points representing two adjacent records will be connected with a horizontal line starting from the left point, and ending above or under the right point.

The end point of the line is then connected with a vertical line to the right point. This method assumes, that value of the given piece of context remains the same until the next record is taken. This method will generate a stairs-graph.

- string data - Visualization of this type will focus on the change of value.

Besides visualization of context data we will make it possible to define *problems* and highlight them on the timeline. We will also make hints on what context values implied the problem.

5.8 Defining problems

A problem can be any unexpected behaviour occurring in an application. Probably the most obvious example is if an application crashing. The problem, however, can also be the wrong reaction on a context change: for example, querying for GPS location data even after the GPS location provider has been disabled.

Erroneous behaviour can be detected either by the user of the application or by the developer during an analysis of the logged data. If a user notices that the application works differently to that as it should be, he should somehow report it. For that very same purpose we will implement a *Big-red-button* feature into the library. It is basically an empty function which can be quickly called by an easy interaction on the UI. This function call can be found in the trace file and it represents an event when a problem has occurred.

Developers being aware of possible erroneous behaviour can define the problems describing such behaviour. When the application reacts incorrectly to a context change, the problem definition should contain the new value of the relevant pieces of context, and the name of the method that describes the wrong reaction.

The problematic value of the piece of context should be defined by a simple condition: a relation to a concrete value. Depending on the datatype of the context, the following relational operators should be used:

- floating point data can be *greater than* or *less than* the specified value

- integer data can also be compared compared with an equality, so besides *greater than* and *less than* operators there can also be used *equals*, *not equals*, *greater than or equal* and *less than or equal*
- string data is not comparable, since lexicographic comparison is irrelevant for us. Therefore a string can be checked for *equality* or *inequality* with a specified value.

The modified Traceview will be able to highlight the defined problems. Big-red-button events and function calls from defined problems will be shown and the value of a piece of context will define the intervals, which will also be highlighted.

5.9 Hints on context

When the problem is defined, its every occurrence is highlighted in the modified traceview application. Since the timestamp of every occurrence is known, the actual values of logged pieces of context can be easily looked up at these moments. From these values we can then presume values of the other pieces of context that may have caused the problem.

6. Implementation

In this chapter we will describe some important parts of the implementation.

6.1 ContextLogger architecture

As discussed in the previous chapter, logging context information needs to be done in a process separate from the application. Communication with this background process is hidden from the application developer inside the `ContextLogger` proxy class. Since every application can have at most one context logger process attached, the proxy class can implement the singleton pattern. Once the class is instantiated and initiated, it is bound to the background service. When started, the service process waits for commands from the application process. Inter-process communication between the proxy class and background service is realized using a Messenger class.

The public interface of the ContextLogger class contains the following methods:

`public static ContextLogger getInstance()`
: returns a reference to the singleton proxy object. It needs to be initialized before it can be used.

`public boolean init(Context)`
: initializes the library: starts the background process and binds it to the context of the application.

`public void addListener(ContextListener)`
: adds the parameter to the list of listeners that will be used during the logging. The parameter can be a predefined listener from our library, or any user-defined class that extends the `DefaultContextListener` class.

`public void clearListeners()`
: clears the list of listeners.

`public void enableTracing(boolean)`

sets whether method tracing should be started. If tracing is disabled, the library saves only context log.

`public void startLogging()`

starts the logging. If tracing is enabled, startMethodTracing() is called in the application's process. Also, sends a message to the background process to start logging.

`public void stopLogging()`

stops the method tracing, if it was started, and sends a message to the background process to stop context logging and save the log file.

`public void stopService()`

stops the background service.

`public void setTraceName(String)`

sets the base filename for trace and log files. Files will be saved in the Downloads folder and the current date-time will be appended to the specified value.

`public void useIntentTarget(boolean)`

sets whether the IntentDataTarget should be used.

`public void useTextFileTarget(boolean)`

sets whether the TextFileDataTarget should be used.

`public void brb()`

an empty function, used for Big-red-button functionality.

6.2 Background process components

The background process of ContextLogger contains three main parts: the `ContextLoggerService` class wrapping around the other parts and responsible for the inter-process communication; the `ContextListeners` capturing the context information; and the `DataManager` responsible for processing the captured information. The architecture of the library is shown in Figure 6.1.

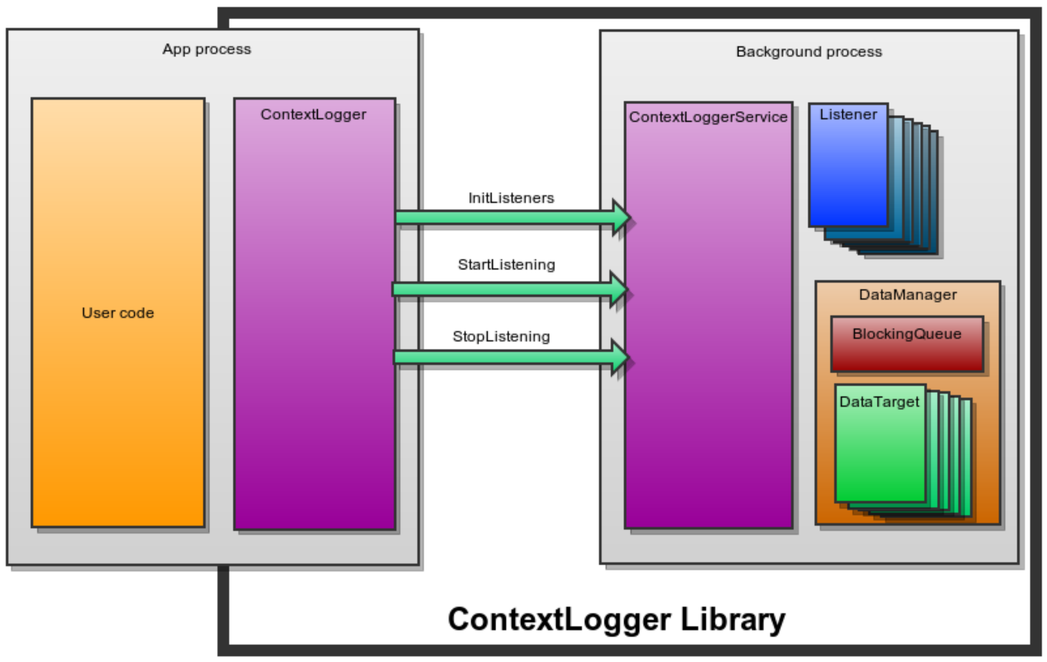


Figure 6.1: Architecture of ContextLogger library

As mentioned in the previous chapter, ContextLogger library needs to take timestamps for log records itself. To make these timestamps as accurate as possible we try to minimize the delay between the context change event and the time sampling. We want to run the event handling code in the listener object as soon as possible after the event actually happened. We achieved this by running each listener in a separate thread and by keeping the code of the listeners' handler function as short as possible.

By using separate threads we make sure that the handler function gets called as soon as the operating system switches to its thread, and no other function blocks his execution. The drawback of having many threads is the overhead of context switching when many of them want to run at the same time. However, if we try to keep the functions short then this overhead will be negligible. Handling functions will only create the timestamp, read the required context values, put all this information into a `LogEntry` object and append it to a queue for further processing. This processing usually consists of expensive I/O operations that would block the calling thread: therefore it is done outside of the listener's thread.

LogEntry is a simple object representing a captured piece of context. It contains a timestamp, a label and the captured context value. LogEntries are produced by the context listeners that insert them into the DataManager's queue.

Entries are then consumed by the DataManager's processing method, which again runs in a separate thread. It takes the first entry from the queue and passes it to the registered `DataTargets`. The queue of entries is an instance of `LinkedBlockingQueue`, which supports blocking operations for inserting and removing elements; therefore, it is ideal for that kind of producer-consumer problem. It blocks the producer if the queue contains the maximum allowed number of elements. In our case the maximum is not specified, it is only bounded by the amount of available memory. When the queue is empty the consumer gets blocked until a new element is inserted.

DataTargets are objects representing the output of the ContextLogger library. The main output is the context log file described in the next section. It is created by the `FileDataTarget`. However, for debugging purposes we have implemented two additional classes. `TextFileDataTarget` generates a similar output as `FileDataTarget` but instead of binary coding of the values in the file everything is saved as plain text. This can be useful during the implementation of additional context listeners. `IntentDataTarget`, instead of saving logs into a file, broadcasts them back to the system. It can be useful if we want to visualise the logs in an application right on the device. Our demo application *CLDemo* uses this method.

6.3 Log file format

Our library will log the code execution trace of the application and the requested context information. For the trace logging we will use the system's built-in functionality: the `startMethodTracing()` function in the `android.os.Debug` class. It generates a log file with a well-defined structure described at [20].

Context data logs will be saved in a separate file. This will also make it possible to process the trace file with the existing debugging tools. The context log file structure will follow a similar structure as the trace logs.

The context log file will be built up from two parts: the key part and the data part. Both of them are created as a separate file at the start of logging. When

log-type value	represented data type
1	int
2	long
3	float
4	double
5	string

Table 6.1: Description of log types

logging stops, the two files are concatenated.

The key part is a plain text file containing two sections. Each section starts with a keyword on a new line, prefixed with a '*'. The two sections are:

version section

contains one line, describing the file version number, currently 1.

logs section

one line for each loggable piece of context. A line consists of three parts, separated by a space: *log-ID* `[space]` *log-type* `[space]` *log-name*.
log-ID is an integer used to identify the piece of context in the data part. Ids in this section do not need to be sorted; however, must form a continuous series, starting with 0.
log-type is an integer value, enumerating the type of the related log records. Possible values are described in Table 6.1
log-name is a string description of the piece of context. It will be used in the user interface.

A key part of an example log file is shown in Code snippet 6.4.

The data part is a binary file created as output from a `DataOutputStream`. It starts with a *header* section followed by a list of *records*. The format of the header section is described in the Code snippet 6.5. Version is currently 1. Start date/time is the output from `gettimeofday()` function call. It is used to synchronize the records of the log file with the records of the trace file.

The format of records is described in the Code snippet 6.6. *LogID* is the identifier of the piece of context, defined in the key part. *Timestamp* is the delta time since the start of the logging. *Value* contains the actual value after the context change. Its type is defined in the Key part.

```
*version
1
*logs
0 3 CPU
1 1 Screen state
2 1 Screen orientation
3 1 Last screen orientation
4 4 Time since orientation change
5 2 Total sent traffic
6 2 Total received traffic
7 2 My sent traffic
8 2 My received traffic
9 5 Wifi connection state
10 1 Wifi strength
11 1 Wifi state
*end
```

Code snippet 6.4: Example key part of the context log file

```
int  magic 0x574f4c53 ('SLOW')
byte version
long start date/time in usec
```

Code snippet 6.5: Example header section

```
int     logID
long    timestamp
logType value
```

Code snippet 6.6: Description of the record structure

6.4 Covered context

ContextLogger library comes with a set of implemented context listeners that captures most of the basic context data. These listeners are the following:

Acceleration listener

SensorEvent-based listener, reads acceleration applied to the device, including the gravity. Logged values are:

- three floats, indicating acceleration applied to the device on the three physical axes relative to the device.
- float, indicating the overall acceleration applied to the device.

The unit of values is m/s^2.

Barometer listener

SensorEvent-based listener, reads from the barometer. Logged value is float, representing the atmospheric pressure in millibars.

Battery listener

BroadcastReceiver-based listener, reads battery stats. Logged values are:

- integer, representing the health of the battery. Value 2 means good health, 3 is for overheating battery, 7 for cold, 4 indicates dead battery, 5 means overcharged, 6 is for unspecified error. If the battery health can not be specified, the logged value is 1.
- integer, representing the level of the battery in percents.
- integer, representing the plugged status of the device: 1 means the device is plugged on an AC charger, 2 is for USB charger, 4 means wireless charger. If the device is unplugged, the logged value is 0.
- integer, indicating whether the battery is present in the device: 1 means the battery is present, 0 indicates no battery.
- integer, representing the battery status: 2 means that the battery is currently charging, 3 means discharging, 4 means not charging, 5 indicates full battery. If the status is unknown, the logged value is 1.

- integer, representing the temperature of the battery.
- integer representing the voltage of the battery.

CPU listener

Periodic listener, reads global CPU usage information from the `/proc/stat` file. The length of the period between reads is 1 second. Returned value is the current CPU usage proportion.

GPS location listener

reads location data from the GPS provider. Logged values are:

- provider status,
- latitude, longitude, altitude,
- bearing,
- speed in m/s, speed in km/h
- number of used satellites.

Minimum distance and minimum elapsed time between updates can be set in the constructor. Default values are 0 meters for distance and 3 seconds for time.

GPS status listener

reads status information of the GPS location provider. Logged value is integer indicating whether the provider is enabled or not, and a string value, containing the enumeration of currently available GPS satellites.

Gyroscope listener

SensorEvent-based listener, reads the rate of rotation of the device. Logged values are:

- three floats, indicating angular speed of the device around the three physical axes, same as used in the acceleration listener. Rotation is considered to be positive, if an observer sitting on the positive infinity on the given axis sees counter-clockwise rotation.
- float, indicating the overall angular speed of the device.

The unit of values is radians/second.

Linear acceleration listener

SensorEvent-based listener, similarly as the acceleration listener, reads acceleration applied to the device, but gravity is subtracted from the values. Logged values are:

- three floats, indicating acceleration applied to the device on the three physical axes relative to the device.
- float, indicating the overall acceleration applied to the device.

The unit of values is m/s^2.

Magnetic field listener

SensorEvent-based listener. The three logged values are floats, representing the force of the ambient magnetic field in the direction of the three axes, same as in the acceleration listener. The unit of the values is microTesla.

Network traffic listener

Reads current network traffic usage. Logged values are integers, representing information such as: the number of sent/received bytes by the system, number of sent/received bytes by the application, number of sent/received packets by the system/application.

Network speed listener

Similarly as the traffic listener, network speed listener logs network traffic information. Instead of saving overall values since the system/application start, the it logs differences between values.

Passive location listener

reads location data from the passive location provider. Logged values are latitude, longitude and the name of the provider the data comes from.

Proximity listener

SensorEvent-based listener. Logged value is float, representing the distance in centimeters of the device from other objects, measured by the proximity sensor. On some devices the sensor returns only a binary '*near*'/'*far*' representation of distance.

RAM listener

Periodic listener for logging information about the memory usage of the application. Logged values are:

- long, representing the available free memory in the system. The unit of the value is byte.
- integer, indicating whether the system considers itself in low memory situation and has started killing background processes. The value is 1 if the situation applies, 0 otherwise.
- long, representing the threshold level of free memory under which the system switches to low memory state. The unit of the value is byte.
- six integers, representing the memory usage of the application process: proportional set size and private heap size on all of dalvik/native/other levels. The units of all values are kilobytes.
- if debug mode is on, six additional integers are logged. These are representing the same information as the previous values, but instead of the application process they describe memory usage of the ContextLogger process. The units of all values are again kilobytes.

Constructor can take two parameters: the first parameter is mandatory, containing the process ID of the application process. It is needed during the memory information lookup. The second parameter is an optional boolean, indicating whether the listener should be run in debug mode. Default value is false.

Rotation listener

SensorEvent-based listener. The four logged values are floats, representing the components of the unit quaternion. Logged values are floats, representing the components of the unit quaternion.

Screen brightness listener

Periodic listener, reads the screen and the soft-key brightness from the `/sys/class/` library. Accurate location of the file containing the context value is heavily dependent on the model of the device and the version of the operating system [27]. Our implementation supports all the test devices. Logged values are integers: screen brightness is represented with a value between 0 and 255. Soft-key brightness can be assigned with values 0 or 1, indicating whether the LEDs are enabled.

Screen orientation listener

Periodic listener, reads current orientation and rotation of the screen of the device. Orientation is represented with an integer value between 0 and 3: 0 stands for landscape orientation, 1 for portrait, 3 for undefined. Value 2 is deprecated. Rotation is also represented with an integer value between 0 and 3, each value stands for a rotation to one edge of the screen. The listener logs also the elapsed time since the last orientation change. This logged value is double.

Screen state listener

BroadcastReceiver-based listener, reads the current power state of the screen. Logged value is an integer: 0 indicates that the screen is turned OFF, 1 for ON.

Telephony Listener

Reads information related to the cell service state of the device. Logged values are:

- integer representing the overall state of the telephony service. Possible values are: 0 for available service (registered to the home operator or roaming), 1 for unavailable service (not registered or currently registering to an operator, registration denied or no signal), 2 for emergency calls available only (registered to an operator, but locked), 3 for turned off radio.
- integer representing whether the device is currently roaming (1 for roaming, 0 otherwise)

- string containing the short name of current registered operator
- string containing the numeric id of the current registered operator
- integer representing whether the device is using GSM (1), CDMA (2), SIP (3) for voice calls, or if there is no radio (0).
- integers, representing the CDMA received signal strength in dBm and Ec/Io value in 10 dB, if the device is using a CDMA radio.
- integers, representing the EVDO received signal strength in dBm, Ec/Io value in 10 dB and signal noise rate, if the device is using a EVDO radio for data transmission.
- integers, representing the GSM signal strength and error bit rate, if the device is using a GSM radio.
- integer representing the current cell data activity: 1 for incoming data, 2 for outgoing, 3 for both directions, 0 for no current activity. Value 4 represents state, when data connection is active, but the underlying physical link is down.
- integer representing the current data connection state: 0 for no connection, 1 for setting up connection, 2 for connected and 3 for suspended state.
- integer representing the call state of the device: 1 for off-hook state, 2 for ringing and 0 for idle.
- integer representing the current data network type. The possible values are listed in Table 6.2.

Temperature listener

SensorEvent-based listener, reads value from the ambient temperature sensor. Logged value is float, representing the temperature around the device in Celsius.

Light listener

1 SensorEvent-based listener, reads the value from the light sensor. Logged value is float representing the ambient light level in lux.

network type value	represented network type
0	UNKNOWN
1	GPRS
2	EDGE
3	UMTS
4	CDMA
5	EVDO_0
6	EVDO_A
7	1xRTT
8	HSDPA
9	HSUPA
10	HSPA
11	IDEN
12	EVDO_B
13	LTE
14	EHRPD
15	HSPAP

Table 6.2: Description of network type values

Wi-Fi listener

BroadcastReceiver-based listener, activated when the Wi-Fi connectivity state was changed. Logged values are:

- string representing the current Wi-Fi state. Possible values are: IDLE, SCANNING, CONNECTING, AUTHENTICATING, CONNECTED, DISCONNECTING, DISCONNECTED, UNAVAILABLE, FAILED.

The following values are available, if the Wi-Fi state is CONNECTED:

- string, containing the BSSID of the currently connected access point, or empty string if no network connected.
- string, containing the SSID of the currently connected network, or empty string if no network connected.
- string containing the IP address.
- integer containing the link speed of the current connection.

Wi-Fi state listener

BroadcastReceiver-based listener, activated when the Wi-Fi option of the device was changed. Logged value is integer, indicating the new state: 1 for disabled state, 2 if the Wi-Fi is currently being enabled, 3 for enabled, 0 for being disabled. Value 4 indicates unknown state which occurs if some error happened during the enabling or disabling.

Wi-Fi RSSI listener

BroadcastReceiver-based listener, activated when the Wi-Fi signal strength has changed. Logged value is integer, representing the new RSSI of the signal in dBm.

6.5 Extensibility

A key objective of the project was extensibility. Since every application can have its own needs of context information, we made it possible for developers to implement their own context listeners. Every class can be used as a context listener, if it extends the abstract `DefaultContextListener` class. Methods which are needed to be overridden are:

`public boolean checkPermissions()`

a function to check whether the application has the correct permissions to access data required by this listener. If every permission is granted, the function should return true, otherwise false. If the listener does not need any special permissions, default implementation can be used, which returns true.

`public abstract void initLogTypes()`

a method for registering the listener. It should call `addLogType(String label, int type)` of DefaultContextListener for every piece of context that the listener is about to cover. The label should contain a simple description of the given piece of context. It will be used for identification during the logging, therefore it should be unique among all the captured pieces. Type describes the type of value we want to save. It needs to be one of DataManager's members: INT, LONG, FLOAT, DOUBLE, STRING. If a piece of context is not registered in this function, it will not be saved in the DataTargets.

`public abstract void startListening()`

a method for starting the listening. It should register the event listeners or BroadcastReceivers that capture the given piece of context from the system. It should also start a new thread that will run the handler method of these objects.

`public abstract void stopListening()`

a method to stop the listening. It should unregister event listeners and BroadcastReceivers used during the logging.

Listeners are instantiated in the application process, then serialized and transported to the background process. Thanks to that, users do not have to change the sources of the library when adding their own listeners.

6.6 Traceview

The original Traceview from the Android SDK is a java application which takes one parameter from the command line: the name of the trace file. If the file is accessible, it creates a `DmTraceReader` instance which is responsible for parsing the trace file and building up the internal data structure from the read records. This reader instance is then passed to the GUI components which need to access the data.

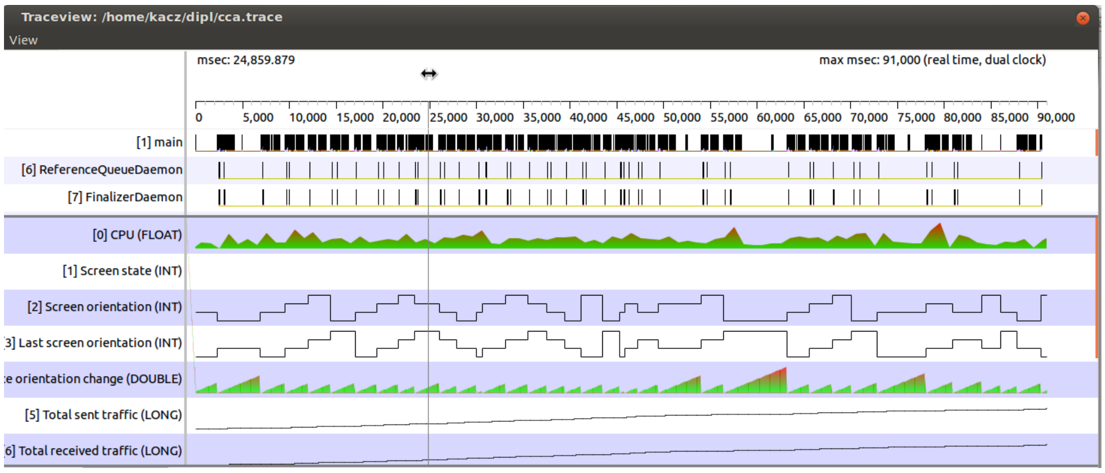
Our modification takes the same parameter and uses it to open both files: trace file with extension .trace added and the context log file with extension .clog. If the trace file is not accessible, the application exits with an error since without it we cannot continue with the analysis. On the other hand, if the log file is missing, we can't do context aware analysis either, but instead of returning with an error, we run Traceview with the original functionality. This makes it possible to use our extended version as a replacement of the original one.

If the log file can be opened, a ContextLogReader instance is created, which is the analog of the trace reader: it parses the log file and makes the data available to the GUI components.

The main components of the Traceview interface are the `TimeLineView`, the `ProfileView` and the `ProblemView`.

TimeLineView in the original Traceview was a highly interactive GUI component visualizing the collected trace data. Records from the trace file were arranged into rows according to the thread they belong to and sorted by the start time. When the mouse moves over the TimeLineView, extra information are displayed about the currently hovered trace record. If the user selects an interval on the timeline, Traceview zooms to the selected interval.

Our modification extends TimeLineView with the ability to visualize trace data (see Figure 6.2) and also achieve the same level of interactivity.



Traceview: /home/kacz/dipl/cca.trace
View
msec: 24,859.879
max msec: 91,000 (real time, dual clock)
0
5,000
10,000
15,000
20,000
25,000
30,000
35,000
40,000
45,000
50,000
55,000
60,000
65,000
70,000
75,000
80,000
85,000
90,000
[1] main
[6] ReferenceQueueDaemon
[7] FinalizerDaemon
[0] CPU (FLOAT)
[1] Screen state (INT)
[2] Screen orientation (INT)
3] Last screen orientation (INT)
e orientation change (DOUBLE)
[5] Total sent traffic (LONG)
6] Total received traffic (LONG)

Figure 6.2: TimelineView with context log visualisation

Mouse-move events convert the mouse position to a corresponding timestamp on the timeline. TimeLineView then checks the value of the actual piece of context at that specific moment. The value of an integer or string type piece of context at a given moment equals the value of the last record before that moment. If the mouse position represents an earlier moment than the timestamp of the first record, the value at that specific time is undefined.

The *value* of a floating point type piece of context at a specific *time* is computed from the value of the two adjacent records by interpolation using the following equation:

$$value = prevValue + \frac{time - prevTime}{nextTime - prevTime} * (nextValue - prevValue)$$

where $prevValue$ and $prevTime$ refers to the last record before the given moment, $nextValue$ and $nextTime$ refers to the first record after it. Again, if the mouse position represents an earlier moment than the timestamp of the first record, or a later moment than the timestamp of the last record, the value at that time is undefined.

On every redraw event the component checks which records fall into the currently visible time interval. For each record the actual coordinates relative to the window coordinate-system are then computed. When all this information is available, the component gradually draws the problem intervals and the highlighted function call, defined in the ProblemView, the visible records and the selected function in the trace panel and the visible part of the graphs on the log panel.

ProfileView component of Traceview remains the same as in the base application, since it is not used during the context analysis.

ProblemView is a new component implemented by us (see Figure 6.3). It consists of two panels: the problem definition panel and the statistics panel.

The problem definition panel allows the user to specify 'problems', i.e. unexpected behaviour of the application. In the scope of this thesis we have implemented two types of problems:

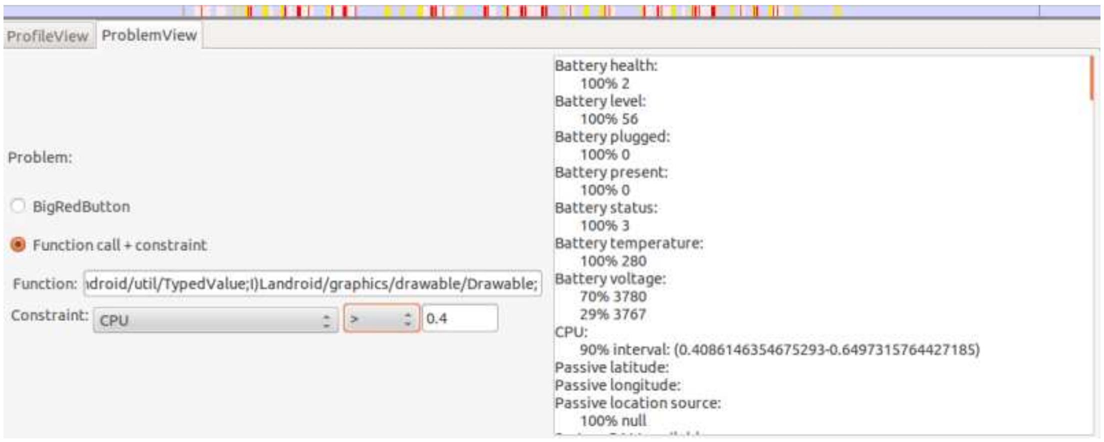
BRB

by selecting this option the Big-red-button signals get instantly highlighted on the timeline and the statistics panel get filled with the computed stats

Context condition + function call

by selecting this option the function definition field and the context condition field become available. By entering a function name we instantly get the function calls highlighted on the timeline. When we also enter a condition for one of the collected piece of context, the timeline highlights the intervals where the condition is met. In this case the highlighted function calls are divided into two groups - the ones which fall into the intervals and the others. The statistics are computed only for the first group.

Computed statistic values depend on the type of the logged piece of context. In every case, the computation is based on the values at the time of occurrences of the defined problem. For discrete and categorical data types (integers, longs and strings) the statistics contain the five most frequent values together with the proportion of their occurrence in the set. For floating point types (floats and doubles) the statistics describe the interval containing 90% of the values by removing the top and bottom 5% of them to exclude outliers.



Problem: BigRedButton [✓ Function call + constraint]

Function: ıdroid/util/TypedValue;I)Landroid/graphics/drawable/Drawable;

Constraint: CPU > 0.4

```
Battery health:
    100% 2
Battery level:
    100% 56
Battery plugged:
    100% 0
Battery present:
    100% 0
Battery status:
    100% 3
Battery temperature:
    100% 280
Battery voltage:
    70% 3780
    29% 3767
CPU:
    90% interval: (0.4086146354675293-0.6497315764427185)
Passive latitude:
Passive longitude:
Passive location source:
    100% null
```

Figure 6.3: Screenshot of ProblemView

7. Use cases and Evaluation

To present the capabilities of our library we have implemented a simple demo application. It has the library connected to it and is able to configure the logging and visualize the collected data.

To show our solution in real situations we wanted to connect our library to an existing application and use it to detect context-aware bugs. We were looking for open source applications with public issue tracking lists. Our goal was to find context-specific bugs on the list, reproduce them and use ContextLogger to identify the actual context responsible for the errors. In the end we have chosen the android version of Firefox web browser.

7.1 Demo application

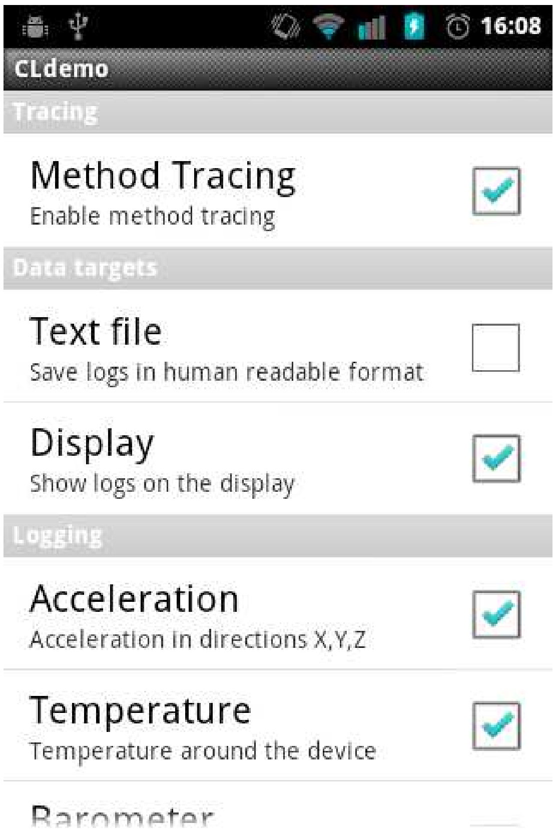
CLDemo is a simple application consisting of one activity. After the startup the screen contains one button labeled *Start*.

To configure the logging process you can open the *Settings screen* by pressing the Menu button (on Android 3.0 and higher by tapping the action overflow icon in the action bar) and selecting the Settings icon.

The Settings screen (shown in Figure 7.1) contains a list of checkboxes for toggling particular features. The first checkbox enables the method tracing. Without this option enabled the library will save only the context log file, and the trace will be not logged. By the next two checkboxes you can configure the used DataTargets. FileDataTarget is always used, as it generates the default output of the logging. *Text file* checkbox will enable TextFileDataTarget to generate a human readable version of the logs. *Display* checkbox enables the IntentDataTarget to broadcast logged context information. This broadcast is captured by the main activity of the application and is then displayed.

Each of the remaining checkboxes represent one contextlistener. By selecting a checkbox you can tell the application to use the given listener during the next logging session.

To start the logging you can to tap the *Start* button on the main activity.



16:08

CLdemo

Tracing

Method Tracing ✓
Enable method tracing

Data targets

Text file ☐
Save logs in human readable format

Display ✓
Show logs on the display

Logging

Acceleration ✓
Acceleration in directions X,Y,Z

Temperature ✓
Temperature around the device

Barometer

Figure 7.1: Screenshot of the Settings screen

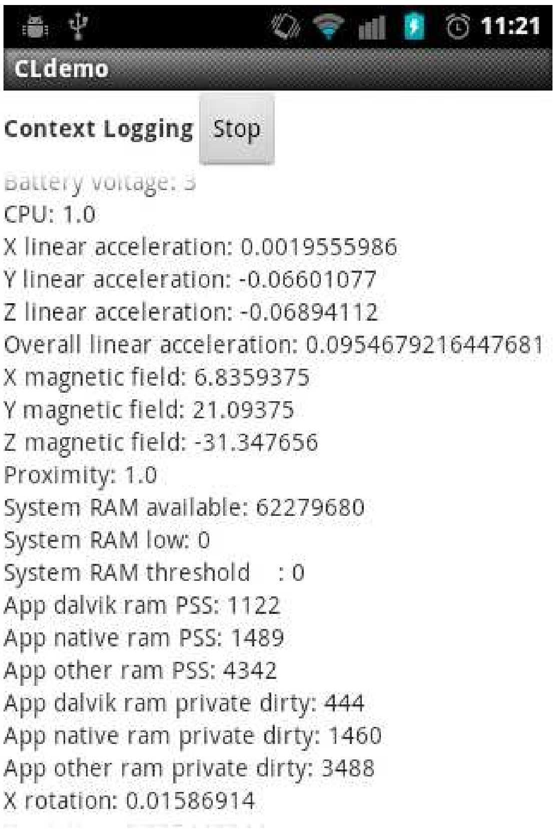
Since an application with an already designed GUI can be hardly modified to contain another button on the interface, logging should be started without the need of adding a GUI element. As a demonstration of this, logging can be started also by pressing the *volume up* button of the device.

After you start the logging session by either way, a Toast message will inform you whether the start was successful or if an error occurred.

After a successful start, if the Display checkbox was selected in the settings menu, the context information will appear on the screen (see Figure 7.2). Each piece of context will be represented by one line containing the label defined in the context listener and the most recently captured value. If a new value is captured, the activity will automatically overwrite the value on the existing line.

The *Big-red-button* functionality is wired to the *volume down* button of the device. When you press it a Toast message will appear with the text '*BRB*'. You can press it several times during the logging.

You can stop the logging by tapping on the Stop button or by pressing the *volume up* button again. After the logging is stopped, the log files are saved



```
Battery voltage: 3
CPU: 1.0
X linear acceleration: 0.0019555986
Y linear acceleration: -0.06601077
Z linear acceleration: -0.06894112
Overall linear acceleration: 0.0954679216447681
X magnetic field: 6.8359375
Y magnetic field: 21.09375
Z magnetic field: -31.347656
Proximity: 1.0
System RAM available: 62279680
System RAM low: 0
System RAM threshold    : 0
App dalvik ram PSS: 1122
App native ram PSS: 1489
App other ram PSS: 4342
App dalvik ram private dirty: 444
App native ram private dirty: 1460
App other ram private dirty: 3488
X rotation: 0.01586914
```

Figure 7.2: Screenshot of CLDemo during a logging session

in the download directory of the system. They can be then transferred to the development machine for analysis.

7.2 Firefox

Mozilla Firefox is an open source web browser developed by the Mozilla Foundation, initially released in 2002. The mobile version, codenamed Fennec, was first released in 2010 for the Maemo operating system and since 2011 the Android system has also been supported. It uses the same Gecko layout engine as the desktop version.

The source code of the browser is stored in a Mercurial repository at address [28]. To build the mobile version, Android SDK and NDK are required.

Firefox project uses Bugzilla as its issue tracker system and the list of known issues is publicly available at address [29]. We have searched through the list for context-aware bugs. We were looking for bugs related to network reliability since as a browser, Firefox requires a stable network connection for correct running. We were also searching general context-aware bugs, related to physical and computing context information, such as screen orientation, battery level, memory usage. Finally we have chosen the bug 739177 [30] to reproduce. To achieve that, first we needed to get a snapshot of the source code from March 25, 2012, the date when the bug was submitted. We have switched to revision [31]. After that we modified the sources by adding code to instantiate our library. We have enabled ContextListeners, which capture relevant context data about the network connectivity and the device in general. The chosen listeners were: WifiListener, WifiRSSIListener, WifiStateListener, TrafficListener, TrafficSpeedListener, BatteryListener, CpuListener, PassiveLocationListener, RamListener, ScreenStateListener, ScreenOrientationListener.

Mozilla has no project in Eclipse, it uses it's own build engine, so we had to also add ContextLogger to the build process. The whole build procedure is described in detail in Appendix B.

We deployed the prepared package to a Nexus 7 device running Android 4.2.2. After running the application for a while we have successfully reproduced the issue. We started the logging and continued with the browsing. When we noticed the expected bad behaviour, we used the BRB function of the library. We ran the logging session for about 90 seconds while browsing. When we stopped the logging, we copied the log files to the development machine in order to analyse them with the extended Traceview.

When opened in Traceview, we could see the trace data and the logged context information visualized. Since we used the BRB function during the logging, we can navigate to the problem definition panel and select the BRB option. This instantly highlights the occurrences of this function call over the timeline, as shown in Figure 7.3. The statistics panel of the ProblemView is also filled with the computed statistics . Most of the values are irrelevant in this case, either because the given piece of context has not changed during the logging and therefore the statistics contain only one value, or because the piece of context is unrelated to the error and therefore many different values occur in the statistics. We can, however, see that at the time of signals the screen orientation was always 'portrait' and 90% of the captured values of the elapsed time since the last orientation change is in the interval (1.096s-1.808s), so the problem always occurred shortly after the orientation changed to portrait. From this information we can conclude that the problem is caused by the portrait screen orientation context. This conclusion corresponds with the issue description.

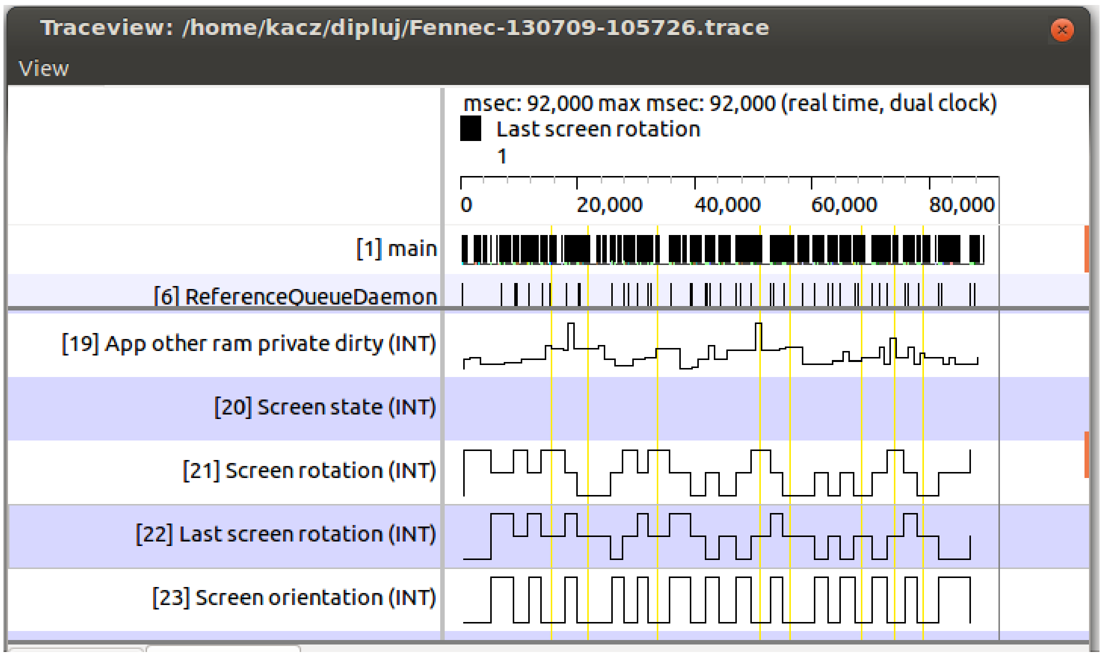

Traceview: /home/kacz/dipluj/Fennec-130709-105726.trace
View
msec: 92,000 max msec: 92,000 (real time, dual clock)
Last screen rotation
1
0
20,000
40,000
60,000
80,000
[1] main
[6] ReferenceQueueDaemon
[19] App other ram private dirty (INT)
[20] Screen state (INT)
[21] Screen rotation (INT)
[22] Last screen rotation (INT)
[23] Screen orientation (INT)

Figure 7.3: Traceview highlighting defined problem

8. Conclusion

The goals of the thesis were to propose a debugging method which takes context information into account by implement a library for collecting such information, and to implement a debug tool for processing the collected information.

The first point was achieved by creating the ContextLogger library. It combines the trace logging supported by the system with a highly extensible context logging, implemented by the library.

The output of the logging is a pair of log files: the trace log and the context log. The trace log file contains records about every method call in the application process. Its structure is defined by the system.

The context log file contains records about the changes in selected pieces of context. The file follows a similar structure to that of the trace file to make the processing easier.

The library has a simple interface which makes it easy to be added on to applications. We have created a simple demo application to demonstrate the functions of the library. We have also added ContextLogger to the mobile version of the Firefox browser and started it while reproducing a context-aware bug. The logged data was then analyzed.

The second goal was achieved by extending the Traceview application from Android SDK. We have added the functionality to visualize logged context information together with the trace data. We have also implemented a problem definition method, where the user can define a problematic combination of a context value and a function call. The defined problem is again visualized and a statistical hint is computed on the context values causing the problem.

We used the extended Traceview application to analyze the logs captured from a real-life application. In particular, we have used for evaluation well known and complex web browser (Firefox for Android) and we have shown on this nontrivial example, how the presented method can be applied to find context dependent bugs.

8.1 Future work

The project should be enhanced in various aspects. First of all, the library should be tried out on more Android applications: in order to learn what further pieces of context are needed and ought to be covered.

Secondly, more comprehensive statistics should be computed from the logged data in order to give more accurate hints on the causing context. One possibility to achieve this is to use log files from several logging sessions instead of only one.

Another way to improve the project is to integrate the context processing with other debugging tools from SDK, such as dmtracedump and the DDMS. This would make it possible to gather new information from the logs and analyze them from a new viewpoint.

Bibliography

[1] Wikipedia: *Ubiquitous computing*
http://en.wikipedia.org/wiki/Ubiquitous_computing

[2] Gartner: *Gartner Says Worldwide Smartphone Sales Reached Its Lowest Growth Rate With 3.7 Per Cent Increase in Fourth Quarter of 2008*
http://www.gartner.com/newsroom/id/910112

[3] Gartner: *Gartner Says Worldwide Mobile Phone Sales to End Users Grew 8 Per Cent in Fourth Quarter 2009; Market Remained Flat in 2009*
http://www.gartner.com/newsroom/id/1306513

[4] Gartner: *Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010*
http://www.gartner.com/newsroom/id/1543014

[5] IDC: *Press release*
http://www.idc.com/getdoc.jsp?containerId=prUS23946013

[6] Gartner: *Gartner Says Asia/Pacific Led Worldwide Mobile Phone Sales to Growth in First Quarter of 2013*
http://www.gartner.com/newsroom/id/2482816

[7] Wikipedia: *App Store (iOS)*
https://en.wikipedia.org/wiki/App_Store_(iOS)

[8] Wikipedia: *Google Play*
http://en.wikipedia.org/wiki/Google_Play

[9] Weiser, M.: *The Computer for the Twenty-First Century*
Scientific American, 1991

[10] Dey, A.K., Abowd, G.D.: *Towards a Better Understanding of Context and Context-Awareness*
In: CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness, 2000

[11] Wei, Edwin JY, and Alvin TS Chan.: *Towards context-awareness in ubiquitous computing*
In: Embedded and Ubiquitous Computing, pp. 706-717. Springer Berlin Heidelberg, 2007.

[12] Android: *Sensors Overview*
http://developer.android.com/guide/topics/sensors/sensors_overview.html

[13] AWARE: *What is AWARE?*
http://www.awareframework.com/home/

[14] FUNF: *Funf Open Sensing Framework*
http://www.funf.org/

[15] Morphone: *A Context-Aware Android-Based Mobile Operating System* http://morphone.elet.polimi.it/

[16] A. Bonetto, M. Maggio, A. Nacci, D. Sciuto, M. D. Santambrogio: *A Context-Aware Android-Based Mobile Operating System: morphone.os* http://www.changegrp.org/doc/morphone_talk_taipei.pdf

[17] Qualcomm Retail Solutions, Inc.: *Gimbal* https://www.gimbal.com/

[18] David Chu, Aman Kansal, Jie Liu, Feng Zhao: *Mobile Apps: It's Time to Move Up to CondOS* http://research.microsoft.com/apps/pubs/default.aspx?id=147238

[19] Android: *Debugging* http://developer.android.com/tools/debugging/index.html

[20] Android: *Profiling with Traceview and dmtracedump* http://developer.android.com/tools/debugging/debugging-tracing.html

[21] MSDN: *Trace Class* http://msdn.microsoft.com/en-us/library/system.diagnostics.trace(v=vs.100).aspx

[22] MSDN: *IntelliTrace* http://msdn.microsoft.com/en-us/library/vstudio/dd264915.aspx

[23] Norbert Ruessmann: *DevTracer – Trace Monitor for Microsoft Windows Phone 7* http://www.devtracer.com/trace-monitor/windowsphone.aspx

[24] Jonathan Levin: *Mac OS X and iOS Internals: To the Apple's Core* Hoboken, N.J. : Wiley, 2012

[25] Android Reference: <*service*> http://developer.android.com/guide/topics/manifest/service-element.html

[26] The Android Open Source Project: *Controlling the Embedded VM* http://www.netmite.com/android/mydroid/dalvik/docs/embedded-vm-control.html

[27] NoDock community: *Input Device and Backlight* https://sites.google.com/site/androidnothize/no-dock/input-device-and-backlight

[28] Mozilla contributors: *Mozilla-central repository* http://hg.mozilla.org/mozilla-central/

[29] Mozilla contributors: *Bugzilla@Mozilla – Components for Firefox for Android* https://bugzilla.mozilla.org/describecomponents.cgi?product=Firefox%20for%20Android

[30] Mozilla contributors: *Mozilla – Bug 739177* https://bugzilla.mozilla.org/show_bug.cgi?id=739177

[31] Mozilla contributors: *Mozilla – Revision 20a01901480f*
http://hg.mozilla.org/mozilla-central/rev/20a01901480f

[32] Wikipedia: *Android System Architecture*
http://en.wikipedia.org/wiki/File:Android-System-Architecture.svg

[33] Steven Sinofsky: *Supporting sensors in Windows 8*
http://blogs.msdn.com/b/b8/archive/2012/01/24/supporting-sensors-in-windows-8.aspx

[34] The Android Open Source Project: *Android 1.6 - Profile.c*
https://android.googlesource.com/platform/dalvik/+/android-1.6_r1/vm/Profile.c

[35] The Android Open Source Project: *Android 2.3 - Profile.c*
https://android.googlesource.com/platform/dalvik/+/android-2.3_r1/vm/Profile.c

[36] The Android Open Source Project: *Android 4.0.1 - Profile.cpp*
https://android.googlesource.com/platform/dalvik/+/android-4.0.1_r1.1/vm/Profile.cpp

List of Abbreviations

1xRTT	Single-carrier Radio Transmission Technology
adb	Android Debug Bridge
AC	alternating current
ADT	Android Developer Tools
API	application programming interface
app	mobile application
BRB	Big-red-burron
BSSID	Basic service set identification
CDMA	Code division multiple access
CDU	Contextual Data Units
CondOS	Context Dataflow Operating System
CPU	central processing unit
DDMS	Dalvik Debug Monitor Server
Ec/Io	measure of evaluation and decisions of CDMA and UMTS
EDGE	Enhanced Data rates for GSM Evolution
EHRPD	Enhanced High Rate Packet Data
EVDO	Evolution-Data Optimized
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUI	graphical user interface
HSDPA	High-Speed Downlink Packet Access
HSUPA	High-Speed Uplink Packet Access
HSPA	High-Speed Packet Access
HSPAP	High-Speed Packet Access Plus
IDE	integrated development environment
IDEN	Integrated Digital Enhanced Network
JDWP	Java Debug Wire Protocol
LED	light-emitting diode
LRU	least recently used
LTE	Long Term Evolution
MIT	Massachusetts Institute of Technology
NDK	Native Development Kit
RSSI	received signal strength indicator
SD card	Secure Digital card
SDK	Software Development Kit
SIP	Session Initiation Protocol
SMS	Short Message Service
SSID	Service set identification
UI	user interface
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
VM	virtual machine
WP8	Windows Phone 8

A. Content of the Attached CD

The attached CD contains the following data:

- `bin/` - Compiled packages of
  - the ContextLogger library
  - the CLDemo application
  - the patched Firefox
  - the extended Traceview
- `doc/` - Javadoc generated for the ContextLogger library
- `log/` - log files captured from Firefox
- `src/` - Source files of
  - the ContextLogger library
  - the CLDemo application
  - the Firefox browser at revision 20a01901480f, patched with Context-Logger
  - a patch file, containing the changes made on Firefox
  - the extended Traceview
- `thesis.pdf` - This document

B. Build instructions

In this chapter we will describe how to build each part of the project. The instructions target the Ubuntu platform. On other distributions they should be very similar.

B.1 Building the ContextLogger library

B.1.1 Prerequisites

In order to build the ContextLogger Library, you need to install Eclipse IDE, Android SDK with API level 14 enabled and Android NDK revision r5c.

B.1.2 Build steps

Extract the contents of the `src/contextlogger.zip` archive from the CD into a empty directory `<sources>`.

To successfully build the project, first you need to compile the native part of the library. You can do that by executing the ndk-build script from the Android NDK in the ContextLogger project library:

```
<sources>/contextlogger$ <ndk-lib>/ndk-build
```

You can now import the ContextLogger project into your Eclipse workspace by choosing *File>Import...* menu and selecting the "*General/Existing projects into Workspace*" option. Then you need to browse the root directory of the project and select *OK* then *Finish*. To create a JAR archive from the library, select the project in the *Package Explorer*, then choose *File>Export...* menu and select *JAVA/JAR* file option. Click *Next* then *Finish*.

B.1.3 Building CLDemo

If you want to build the demo application, extract the contents of the `src/demo.zip` archive from the CD into the `<sources>` directory and import the CLDemo project into your workspace. If you have connected your device, click Run to execute immediately the application.

B.2 Building Traceview

B.2.1 Prerequisites

In order to successfully build the extended Traceview application you need to download the sources of three projects from the Android SDK. Create an empty directory `<sdk-dir>`. You can download the sources by running commands:

```
<sdk-dir>$ git clone \
    https://android.googlesource.com/platform/tools/swt tools/swt
<sdk-dir>$ git clone \
    https://android.googlesource.com/platform/prebuilts/tools \
    prebuilts/tools
<sdk-dir>$ git clone \
    https://android.googlesource.com/platform/tools/base tools/base
```

B.2.2 Build steps

1. Import project to your Eclipse workspace from `<sdk-dir>/tools/base/common` directory.
2. Add variable to the classpath: select the common project in the *Package Explorer* then choose *File>Properties*. On the left side select *Java Build Path* and open the *Libraries* tab. Click the *Add variable...* button. In the new window click *Configure variables...* and then *New...*. Enter `ANDROID_SRC` to the name field and `<sdk-dir>` into the *Path*. Click *OK* to exit the dialogs.
3. Import project from the `<sdk-dir>/tools/swt/sdkstats` directory.
4. Add classpath variable `ANDROID_OUT_FRAMEWORK` with value `<sdk-dir>/prebuilts/tools/<platform>/swt`.
5. Import the ContextLogger project from `<sources>/contextlogger` directory.
6. Extract the contents of the `src/traceview.zip` archive into the `<sources>` directory and import the traceview project into the workspace.
7. Select Traceview in the *Package Explorer* and click *File >Export...*. Choose *Runnable JAR file* and navigate through the wizzard.

To run Traceview use the following command:

```
<export-dir>$ java -Xmx1600M -jar traceview.jar <trace-name>
```

B.3 Building Firefox

B.3.1 Prerequisites

To successfully build the chosen revision of Firefox you need to complete the following prerequisites:

1. Install JDK version 6 or newer
2. Install Gecko Requirements by running the following commands:

   ```
   sudo apt-get install mercurial ccache
   sudo apt-get build-dep firefox
   ```

3. Install Android NDK revision 4 or newer, except revision 7, which is not working

```
wget http://dl.google.com/android/ndk/android-ndk-r5c-linux-x86.tar.bz2
tar -xjf android-ndk-r5c-linux-x86.tar.bz2
```

4. Install Android SDK

```
wget http://dl.google.com/android/android-sdk_r22.0.1-linux.tgz
tar -xzf android-sdk_r22.0.1-linux.tgz
./android-sdk-linux_x86/tools/android update sdk -f -u
./android-sdk-linux_x86/tools/android update adb
```

If all the prerequisites are completed, you can get the Firefox source code. You can either clone the mozilla-central repository into `<moz-dir>` and then switch to revision 20a01901480f:

```
hg clone http://hg.mozilla.org/mozilla-central/ <moz-dir>
hg update -r 20a01901480f
```

or immediately download the sources of the given revision from address [31].

When you have the sources, first you need to configure the build process. To do that, create a text file `mozconfig` in your `<moz-dir>` directory and add the following lines to it with the correct values inserted:

```
# Add the correct paths here:
ac_add_options --with-android-ndk="<ndk-dir>"
ac_add_options --with-android-sdk="<sdk-dir>/platforms/android-16"
ac_add_options --with-android-version=5

# android options
ac_add_options --enable-application=mobile/android
ac_add_options --target=arm-linux-androideabi
ac_add_options --with-ccache
ac_add_options --enable-tests

mk_add_options MOZ_OBJDIR=./objdir-droid
mk_add_options MOZ_MAKE_FLAGS="-j9 -s"
```

After the file is created, export an environment variable containing its path:

```
export MOZCONFIG=<moz-dir>/mozconfig
```

At this point you are ready to build the selected revision of Firefox by following commands:

```
<moz-dir>$ make -f client.mk
<moz-dir>$ make -C objdir-droid package
```

B.3.2 Import of ContextLogger

To add ContextLogger to Firefox, you need to copy the ContextLogger.jar file into the `<moz-dir>/libs` directory.

B.3.3 Change the code

When the jar file is added, the following changes are required to use the library in the application:

1. Add instantiation of the ContextLogger class to the GeckoApp.java file:
   (a) Create private variables in the GeckoApp class to hold the reference to the ContextLogger instance.
   (b) Instantiate the library in the onCreate() function of GeckoApp.
   (c) Add key event handler function for volume keys to start/stop the logging.
   (d) Stop the library in the onDestroy() function of GeckoApp.
   (e) Add the required imports to the beginning of the file.
2. Change the manifest file of the application, AndroidManifest.xml.in:
   (a) Add user permission required by the listeners.
   (b) Add entry point for the background service with option to run in a separate process.
3. Add ContextLogger.jar to the classpath and to the input of the dx tool in android/base/Makefile.in file.
4. Tell the packager to include libtimesource.so from the ContextLogger.jar into the package.

All these changes are included in the patch file on the attached media. The sources of Firefox with the patch applied is also available on the CD.

When the changes are made, you can build the commands mentioned above. Result `Fennec-14.0a1.en-US.android-arm.apk` package can be found in `<moz-dir>/objdir-driod/dist` directory. You can copy it to your device to install.