Introducing Application-Specific Communication Channels in myHealthAssistant

Einführung anwendungseigener Kommunikationskanäle in myHealthAssistant

Bachelor-Thesis
Jens Sauer

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Datenbanken und Verteile Systeme (DVS)

Jens Sauer
Studiengang: Bachelor Wirtschaftsinformatik

Bachelor-Thesis
Thema: "Introducing Application-Specific Communication Channels in myHealthAssistant"

Thema: "Einführung anwendungseigener Kommunikationskanäle in myHealthAssistant“

Eingereicht: 24. Mai 2013

Betreuer: Prof. Alejandro Buchmann und Christian Seeger

Department of Computer Science
Databases and Distributed Systems Group
Hochschulstraße 10
64289 Darmstadt
Germany

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Sämtliche aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Darmstadt, den 24. Mai 2013

Contents

1 Introduction

Communication channels connect one or several data sources to one or several data sinks. The term is usually used in telecommunication and computer networks. Communication channels refer either to the physical transmission over a medium such as a wire or to a logical connection where more than one channel is transmitted over the same medium, for example radio channels. In computer systems the physical part is usually hidden for developers and the focus is on logical connections within and between applications. Open communication channels or broadcasts usually use a one-to-many relation and transmit a signal that any customer can receive. In satellite television for example, the transmission is fast and has a wide range of customers but it is difficult to exclude receivers and to verify whether a transmission was successful. In order to exclude receivers from reading sensitive data, extra effort like, data encryption for pay TV, can be used.[1]

Application-specific communication channels secure sensitive data by limiting the receivers that are able to receive the message. In postal delivery for example, a letter may be addressed to a company, house, family or a person, which is more efficient than encrypting the letters and making it available to all receivers, whereas only the correct receiver can decrypt it. In addition, with more effort it can be verified that a letter was received by using a registered mail which the recipient has to sign upon reception. Using communication channels, there are always trade-offs between different characteristics and performances. Examples are: system and channel utilization, delivery time, security, addressability of receivers and others.

MyHealthAssistant is a middleware that manages a variety of sensors, collects their readings (event types) and publishes them to interested applications. The middleware has an event driven architecture that reacts to significant state changes, such as incoming sensor readings, new sensor connections and others.[2] The collected sensor readings are forwarded to an open communication channel to which every application can register for and receive transmitted events. The Android operating system supports open communication channels by a broadcast message passing system. This is an efficient (low system utilization, high delivery ratio, and

[1] Shannon et al. (1948)

[2] Seeger et al. (2012)

low delivery time) way to inform a variety of third party applications but has drawbacks in securing sensitive data and restricting access for different event types.

> *"Improvements and new developments of sensor nodes have opened many new opportunities in wireless sensor networks. As new sensor types arise, and as existing sensors become more powerful, more accurate and more energy efficient, those advancements enrich the sensing capabilities and widen the range of possible applications."*[3]

A growing variety and quality of sensors and applications leads to new challenges for myHealthAssistant. More sensitive data like insulin levels, blood pressure or heart attack warnings can be available by using smaller, more energy efficient and more precise sensors. Access to this sensitive data should be restricted to approved applications.

A growing variety of sensors will lead to a lot more event types to be handled by myHealthAssistant. In terms of energy savings the middleware should be able to decide if a sensor is not used by any application and stop it, as well as starting a sensor when an application requests events from a stopped sensor. Another use case could be to tell a sensor to reduce or increase its sampling rate in order to save energy or provide more detailed information.

In order to support these decisions myHealthAssistant needs more knowledge about applications and the event types they are interested in (subscribers) and an overview of all available sensor (publishers). The knowledge about subscribers and publishers provides the necessary information for a Transformation Manger that decides if a requested and not available event type can be transformed using a subset of the available event types. For example an ECG stream contains the information about the heart rate, but a transformation has to be done to create heart rate events out of ECG streams.[4]

The main task of this thesis is to design and implement a solution that faces these challenges and satisfies the need for restricted access to event types/channels, more sensor management functionality and information needed for supporting event transformations. After the design and implementation, a prototype has to be evaluated with respect to the performance impact

[3] Seeger et al. (2011)

[4] Seeger et al. (2013)

of application-specific communication channels on myHealthAssistant. The direct comparison with open communication channels clarifies how the managing overhead and the application-specific communication channels affect the behavior of the over-all system.

2 Android Communication Overview

Android has a complex communication system which provides a rich inter-application message passing system that encourages inter-application collaboration and reduces developer burden by facilitating Component reuse.[5] The more complex message passing system and the lack of a clear distinction between inter- and intra-application communication lead to a confusion by the developers and numerous vulnerabilities in third-party applications.[6]

In this Chapter, basics of the Android security model regarding the communication and the logical building blocks of applications (Components) are shown. Vulnerabilities of the Android communication system and tips for developers how to avoid them are shown in Section 3.

Android’s security model is unique and quite different from the standard desktop principle.[7] Each application on an Android device runs in its own virtual sandbox. By default, the Android system assigns each application a unique Linux user ID (the ID is used only by the system and is unknown to the application) unlike desktop applications that usually run under the same user ID. Running under the same user ID applications share the rights a user has and can access the same data as the user. On Android each application has its own ID and can only access its own files by default.

This principle of least privilege where applications only have access to the Components they require to do their work and not more, creates a secure environment in which an application cannot access parts of the system for which it is not given permission.[8] Isolated applications can use the rich inter-application message passing system to encourage inter-application collaboration. For example a free gaming application should not be able to access user data of

[5] Chin et al. (2011)

[6] Kantola et al. (2012)

[7] Burns (2009)

[8] Android (2013), http://developer.android.com/guide/components/fundamentals.html, 28-March-2013

a highly trusted banking application[9], but it might be useful for the gaming application to communicate with a social network app in order to post achievements. In the next chapters the essential building blocks of an application and the communication between them (Intents) are explained.

2.1 Intents

Intents are messages for the communication between essential building blocks within an application (intra-application communication) as well as messages between different applications (inter-application communication). Intents have a lot of implementation details[10], but the basic idea is that they represent serialized data. This data can be passed as a message with optional information.[11]

There are explicit and implicit Intents. An explicit Intent specifies a Component to which it should be delivered to, whereas an implicit Intent requests delivery to any Component that supports a desired operation. "Its most significant use is in the launching of Activities, where it can be thought of as the glue between Activities."[12] For example, the user clicks on the street address of a contact and expects it to be opened in an application that displays the location on a map. The system sends an Intent containing the street address to a map application and the address is displayed on a map. This can be achieved by sending an explicit Intent to the Google Maps application or by sending an implicit Intent that would be delivered to any application that declares it provides mapping functionality (e.g. Yahoo! Maps, Bing Maps or Google Maps). The delivery process (Intent resolution) for explicit and implicit Intents is described in the sections 2.1.2 and 2.1.1.

The Android system tries to find the appropriate Activity, Service or set of broadcast receivers to respond to the Intent, creating an instance if necessary or contact the user if more Activities with the same priority match the Intent. There is no overlap within these messaging systems. Broadcast Intents are delivered only to broadcast receivers, never to Activities or Services.

[9] Chin et al. (2011)

[10] Android (2013), http://developer.android.com/reference/android/content/Intent.html, 28-March-2013

[11] Burns (2009)

[12] Android (2013) http://developer.android.com/reference/android/content/Intent.html, 08-April-2013

Intents are also used by the operating system as notifications about the system state, such as battery changes or that the system has finished booting.[13]

The primary and some secondary attributes and their meanings are shown in Table 1. Intent attributes are important for the implicit and explicit delivery process shown in the next sections.

Primary attributes	Description
Action	The general action to be performed, such as ACTION–VIEW, ACTION–EDIT, etc.
Data	The data to operate on, such as a person record in the contact database
Secondary attributes	**Description**
Category	Gives additional information about the action to execute
Type	Specifies an explicit type of the Intent data. Normally a type is inferred from the data itself. By setting this attribute, the evaluation is disabled and an explicit type is forced.
Component	Specifies the explicit name of a Component class to use for the Intent
Extras	A Bundle of any additional information. For example extra pieces for an e-mail, like a subject, body, etc.

Table 1: Primary and Secondary Attributes of an Intent[14]

An example for an implicit Intent would be `Intent(action: buy, data: milk)`. Now the Android system tries to find all installed applications/Components that provide the functionality to buy milk.

[13] Chin et al. (2011)

[14] Android (2013), http://developer.android.com/reference/android/content/Intent.html, 02-May-2013

2.1.1 Implicit Intents

Implicit Intents do not name a target (the Component attribute name is blank). In the absence of a designated target, the Android system has to find the best Component/Components to handle the Intent.

The best metaphor would be a conversation, where someone tells you (Android system) what he wants to do (implicit Intent) and you tell him where to go to fulfill the desired task (Component). Components define and tell the Android system what actions, types and categories they support by using Intent filters in the basic configuration file, the AndroidManifest.xml, of each application.[15]

Three attributes of an Intent are used for the implicit resolution, the action-, data/type[16]- and category-field, shown in Table 2.

Attribute	Description
Action	If set, the Component has to support that action.
Data/Type	If set or inferred from the data attribute, the Component has to support that specific type. The type can also be a scheme such as *http:* or *mailto:*.
Category	Zero or more categories can be set to describe more detailed the desired action to perform. A Component has to support all set categories in order to receive the Intent. By only supporting one of two set categories the Intent will not be delivered.

Table 2: Attributes used for implicit Intent resolution[17]

Referring to the example where a user wants a street address shown on a map using the implicit Intent resolution may lead to three Activities (Yahoo! Maps, Bing Maps or Google Maps) that support the desired functionality. By default the system displays all matching Activities to the user and waits for a user to choose one. If the user already set a default Activity to show a street address on a map, the system will always pick the Activity specified by the user. It is similar to the use of a default internet-browser for all http links on a PC.

[15] Dynamically defined broadcast receivers register themselves at runtime with one or more Intent-filters.

[16] If the type field is not set it gets inferred by the data field.

[17] Android (2013), http://developer.android.com/reference/android/content/Intent.html, 16-May-2013

If an Intent tries to implicitly start a Service and more than one Service supports the desired functionality, the user will not be asked to pick one, instead the system will choose one.[18] Broadcast Intents are delivered to all matching receivers.

Implicit Intent resolution encourages inter-application collaboration, but also has some drawbacks when not used carefully. Security risks are shown in Chapter 3.

2.1.2 Explicit Intents

Explicit Intents can be seen as a command to go to a specific place. Like a mother telling her child to go to school. The child is forced to go to school even if it does not want to do things that a school provides. The child wants to play laser tag, but the school only provides things like learn math. If the child would use implicit Intent resolution it would be sent to a laser tag arena. But for explicit Intent resolution the child is forced to go to school, discarding what it actually wants to do.

In a more formal way: Explicit Intents are only delivered to an instance of the designated target class. The only thing that matters is the Component (explicit address in Android), discarding all other fields of an Intent. Component names are generally unknown to developers of other applications and they are typically used for a secure intra-application messaging.[19]

"An explicit Intent is guaranteed delivered to the intended recipient."[20]

An Intent can be explicitly addressed by using one of the following methods *setComponent()*, *setClass()* or *setClassName()*. These three methods set the secondary attribute (component) of an Intent. When the component attribute field is set all other fields become optional.[21]

A *ComponentName* contains two pieces of information: the package[22] (a String) it exists in, and the class (a String) name inside of that package.[23] Referring to the Google Maps example the code for explicit addressing is:

[18] Kantola et al. (2012)

[19] Android (2013), http://developer.android.com/guide/components/intents-filters.html, 28-March-2013

[20] Chin et al. (2011)

[21] Android (2013), http://developer.android.com/reference/android/content/Intent.html, 20-April-2013

[22] Every Android application has a unique package name. The concept comes from Package Naming Conventions in Java and uses a reverse domain name scheme. Because most (all) publishers own a unique domain name.

```
intent.setComponent(new ComponentName("com.google.android.apps.maps",
                                "com.google.android.maps.MapsActivity"));
```

Another interesting method is *setPackage()*. It combines explicit addressing with an implicit Intent resolution. The method sets an explicit application package name that limits the range of Components this Intent will resolve to. If not null, the Intent can only be implicitly resolved to Components that match the given application package name.[24] In the maps example the code would be:

```
intent.setPackage("com.google.android.apps.maps");
```

The implicit Intent resolution would start in the Google Maps application and only resolve to Components that are declared in this application. For example a newspaper delivery, an explicit Intent would be delivered to the specified person in the flat only. Using *setPackage()* would deliver it to the mailbox. Within the boundaries of a flat, the newspaper gets delivered to a person that declared it is interested in reading it.

2.1.3 Intent-Filters

Intent filters are the opposite of declaring what an Intent wants to do. It specifies what a component can do and what action, type/data and category attributes it supports. It is used to inform the system which implicit Intent it can handle. Activities, Services and broadcast receivers can have one or more Intent filters, whereas Content Providers cannot receive Intents and cannot declare Intent filters.

Intent filters can be used to filter out unwanted implicit Intents. Explicit Intents are not affected by Intent filters and will always be delivered.[25] If a component is available to other applications, the component has to be able to receive and process explicit Intents, otherwise unintended behavior as well as security issues may occur. In Section 3.1.1 it is explained how a component can be made available to other applications.

For example the `Intent(action: steal, data: milk)` explicitly addressed to a store-application will pass the Intent filters. But Store-applications do not (want to) provide the stealing action at all. The application has to deal with customers/Intents with bad intentions,

[23] Android (2013), http://developer.android.com/reference/android/content/ComponentName.html, 09-April-2013

[24] Android (2013), http://developer.android.com/reference/android/content/Intent.html, 28-March-2013

[25] Android (2013), http://developer.android.com/guide/components/intents-filters.html, 09-April-2013

that know the market's address and ignore no-stealing signs on the entry door (Intent filters). This case shows the need to always check incoming Intents and that relying on Intent-filters is not sufficient for security.

2.2 Components

Applications Components are the essential building blocks of an Android application. Each Component has its own specific role.

Activities are used for the visual interface. Services are running in the background doing long running operations without blocking the user interface. Broadcast Receivers are used to receive and forward information. Content Providers are used for a persistent internal data storage as well as a mechanism for sharing information between applications. Each one is unique with its distinct lifecycle that defines how the component is created and destroyed. The combination of Components helps to define an applications overall behavior.[26]

2.2.1 Activity

An Activity is a single, focused thing that a user can do. Activities are in almost all cases used to interact with the user, the Activity class takes care of creating a window in which the user interface can be placed.[27] An Activity represents a single screen/window with a user interface; in general all visible portions of applications are Activities.

Activites are started with Intents, and they can return data to their invoking Components upon completion.[28] For example, an email application might have three Activities. The first Activity shows a list of new emails, the second is used to compose an email and the third is an Activity for reading emails. All Activities are independent from each other, but the Intent communication creates a cohesive user experience within an email application by seamlessly switching between these Activities. Each Activity can be started (if allowed) by another application. For example a camera application can start the Activity in the email application that composes new email in order to share a taken picture with friends.[29]

[26] Android (2013), http://developer.android.com/guide/components/fundamentals.html, 09-April-2013

[27] Android (2013), http://developer.android.com/reference/android/app/Activity.html, 21-April-2013

[28] Chin et al. (2011)

[29] Android (2013), http://developer.android.com/guide/components/fundamentals.html, 09-April-2013

Activities are started with Intents and have their own lifecycle and state paths through it, shown in Appendix 1. A developer has to be aware of the different states of an Activity in the lifecycle in order to avoid using unnecessary system resources when the application is not displayed.

2.2.2 Service

Services run in the background and are used to perform long-running operations or do work for a remote process. A Service does not interact with a user, it does not provide a user interface.[30] It can be seen as a silent worker in the background. For example a Service is used to play music in the background, while the user is using an Activity of a different application. Another common task is fetching data over the network without blocking user interaction within an Activity.[31]

Services are started with Intents using the method *startService()* or *bindService()*. When a Service is created with *startService(),* it is running until is stopped by itself or a client. Other Components can use *bindService()* which lets the binder invoke methods that are declared in the targets Service's interface.[32] A bound Service is stopped when all bound Components unbind by calling the *unbindService()*. Both lifecycles are shown in Figure 1.

[30] Chin et al. (2011)

[31] Android (2013), http://developer.android.com/guide/components/fundamentals.html, 09-April-2013

[32] Chin et al. (2011)

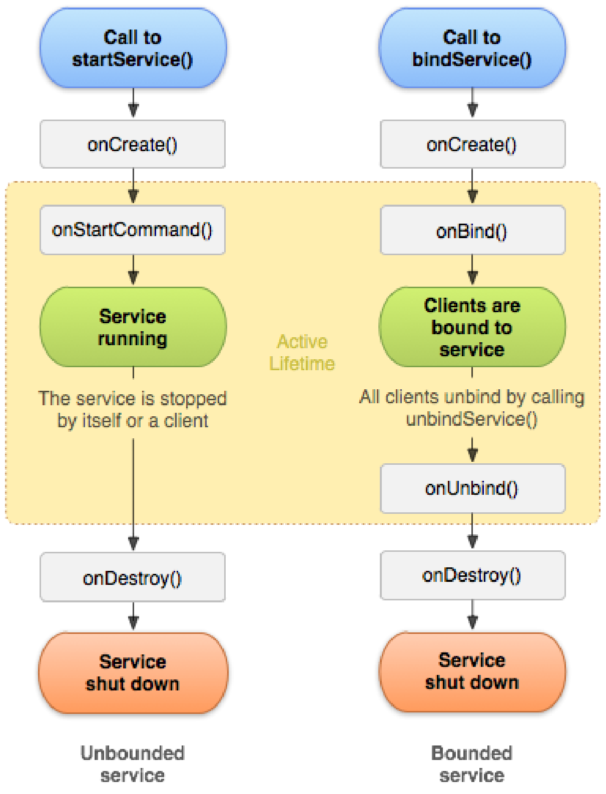


Figure 1: Lifecycle of a Service[33]

2.2.3 Broadcast Receiver

The basic principle of broadcast receivers is that the sender describes what kind of information the Intent contains and every interested broadcast receiver (declared through Intent filters) receives the Intent. Receivers are triggered by the reception of a matching Intent and then run in the background to handle it. Typically Broadcast Receivers are short-lived and often relay messages to Activities or Services.

There are three types of broadcast receiver Intents: Normal, Ordered and Sticky. Normal broadcasts are sent to all registered receivers at once and disappear after delivery. An Ordered broadcasts is based on a priority rating. The highest priority receiver can modify or stop the propagation of the broadcast before it is delivered to the second highest priority receiver and so on.

[33] Android (2013), http://developer.android.com/guide/components/services.html, 09-April-2013

Sticky broadcasts stay available after they have been delivered and are re-broadcasted to future Receivers.[34] For example; the last state of the energy status of the phone is saved as sticky broadcast, this way every application can request the last state of the energy status even after the initial event happened. Security risks for Broadcasts are explained in Chapter 3 Android Communication Security, the basic Android features to secure broadcast senders and receivers are explained in Section 3.1.2 Android Permissions.

2.2.3.1 Static Broadcast Receiver

Static receivers and their Intent filters are declared in the application's AndroidManifest.xml. The AndroidManifest.xml presents essential information about the application to the Android system, information the system must have before it can run any of the application's code.[35]

Static Receivers can be reached by implicit and explicit Intents. In Section 3.1.1 Exporting a Component the difference between a private (can only receive broadcast within the same application) and exported (can receive broadcasts from all applications) is described in more detail.

An example task for a Static receiver is to start a Service as soon as the Android system booting is finished. The Android system sends a system broadcast when it is done booting[36]. After receiving this broadcast the static receiver will start their applicable services and then terminate. In Figure 2 it is shown that two broadcasts are received and the Static receivers get instantiated twice, runs its code reaches the end of its lifecycle. After the end of its lifecycle the Static receiver instance waits to be collected by the garbage collector.

[34] Chin et al. (2011)

[35] Android (2013), http://developer.android.com/guide/topics/manifest/manifest-intro.html, 09-April-2013

[36] System broadcast Intent uses the action: "android.intent.action.BOOT–COMPLETED"

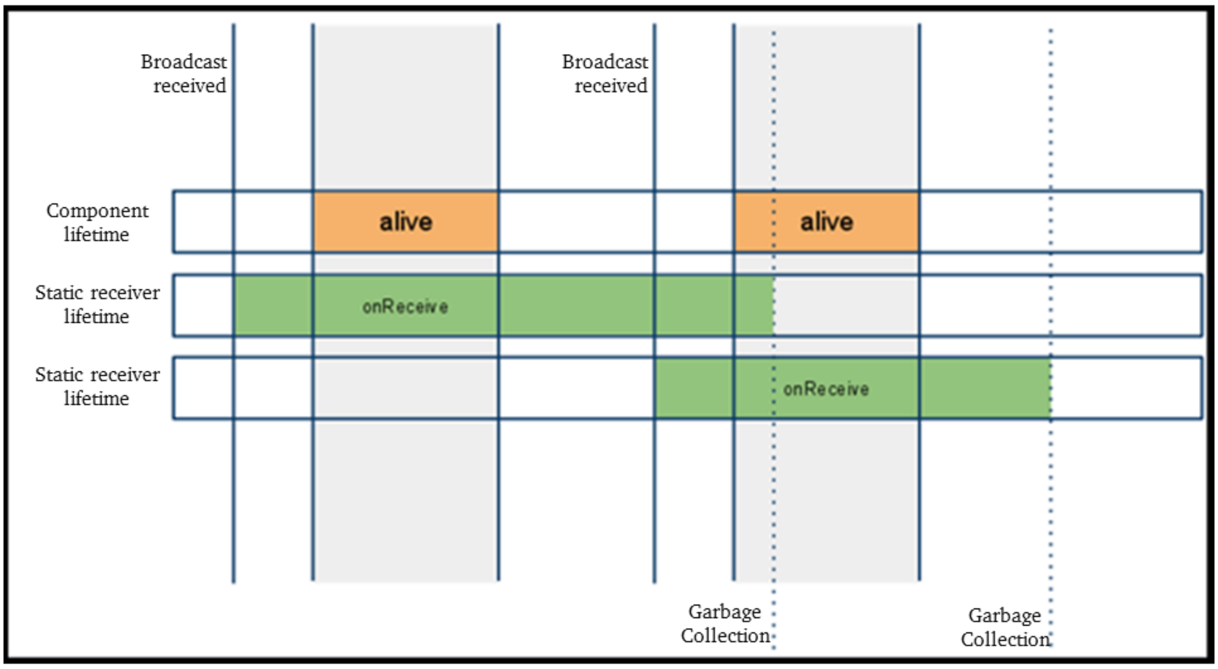


Figure 2: Lifecycle of a Static broadcast receiver[37]

2.2.3.2 Dynamic Broadcast Receiver

A Dynamic Broadcast Receiver is declared at runtime. The main difference between a dynamic and a static receiver lies in the lifecycle. While a static receiver lifecycle ends with its complete task, a dynamic receiver's lifecycle is bound to its component lifecycle. For example; if a receiver is registered at the *onCreate()* method of a Service and unregistered at the *onDestroy()* method it has the same lifecycle as the Service, shown in Figure 3. Although the Dynamic broadcast receiver is not able to receive explicit Intents:

> *"Dynamic Receivers are an exception, since some can only receive implicit Intents. Thus, they are always public[38], and Intents they receive must match one of their Intent Filters."*[39]

Dynamic broadcast receiver cannot be addressed explicitly.

[37] Devmaze (2013) http://devmaze.wordpress.com/2011/07/17/android-components-lifetime/, 09-April-2013

[38] They can receive broadcasts from all applications.

[39] Kantola et al. (2012)

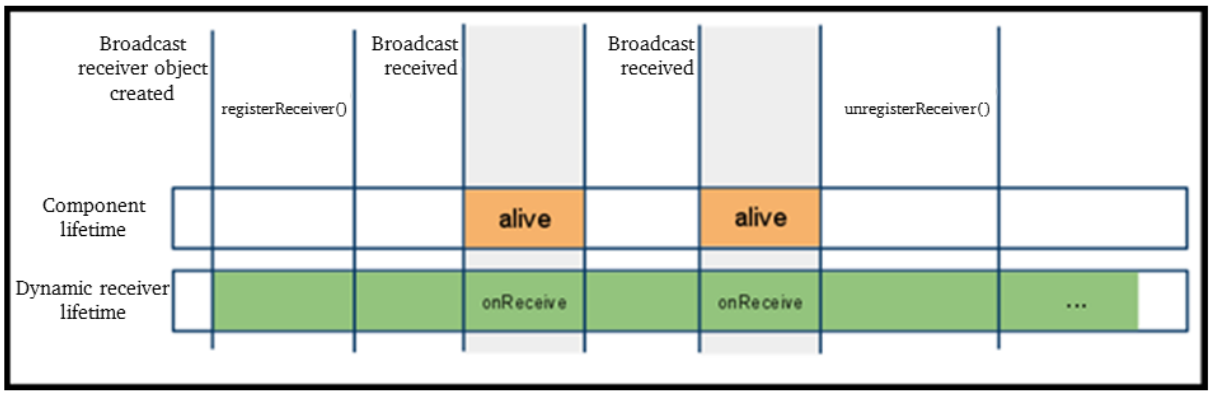


Figure 3: Lifecycle of a Dynamic broadcast receiver[40]

2.2.3.3 Local Broadcast Receiver

A Local Broadcast receiver helps to register for and send broadcasts of Intents to local objects within its application. This has a number of advantages over sending global broadcasts with *sendBroadcast()*.

- There is no risk about leaking private data to other applications, because application boundaries cannot be passed (no inter-application communication).
- There is no risk about security holes that could be exploited, because other applications cannot sent broadcast to the receiver (no inter-application communication)
- It is more efficient than sending a global broadcast through the system.

Local broadcast receivers should always be used for intra-application communication due to the big advantages compared to a regular broadcast receiver.[41]

2.2.4 Content Provider

A Content provider manages access to a central repository of data. It presents data to external applications as one or more tables that are similar to the tables found in a relational database.[42]

[40] Devmaze (2013), http://devmaze.wordpress.com/2011/07/17/android-components-lifetime/, 09-April-2013

[41] Android (2013), http://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html, 09-April-2013

[42] Android (2013), http://developer.android.com/guide/topics/providers/content-provider-basics.html, 14-May-2013

A Content provider is used for persistent internal data storage as well as a mechanism for sharing information between applications.[43] Content Providers are different from the other three Components, they cannot receive Intents.

This component is not explained in detail, because a shared data storage does not fit the concept of an event-driven architecture. Hence it is not suitable for a solution approach for myHealthAssistant.

3 Android Communication Security

Problems in the virtual sandbox arise when the Android API pokes holes in the sandbox by allowing applications to communicate with another.[44] The Intents used for message passing can be an application attack surface. The content of messages can be sniffed, modified, stolen, or replaced which can compromise user privacy.

The attack surface could be the implicit Intent resolution, because a Component can claim to handle any action, type, or category, regardless of whether it is actually designed for the desired operation.[45]

The rich inter-application message passing system that encourages inter-application collaboration reduces the burden for the developers by facilitating Component reuse. But at the same time it is a common developer mistake to expose a Component or message unintentionally to third-party applications. This can lead to unnecessary exposure of internal application messages and Components. The missing distinction between intra- and inter-application communication confuses some developers and lead to unnecessary exposure of internal application messages and Components.[46] An example for a malicious mobile phone application is the SMS Message Spy Pro which disguises itself as a tip calculator and at the same time forwards all sent and received messages to a third party.[47,48]

[43] Chin et al. (2013)

[44] Kantola et al. (2012)

[45] Chin et al. (2011)

[46] Kantola et al. (2012)

[47] Chin et al. (2011)

[48] Vennon (2010)

By default, applications do not have the ability to interact with sensitive parts of the system API. However, the user can grant an application additional permissions during installation. In Section 3.1.2 Android Permissions the concept and drawbacks are explained. The focus of this Chapter is on securing applications from each other, not considering attacks on the Android operating system.

3.1 Component Declaration

A component must be declared in the application's AndroidManifest.xml to receive Intents.[49] This is the configuration file that accompanies the application during installation. The AndroidManifest.xml presents essential information about the application to the Android system. The system must have this information before it can run any of the application's code.[50] Developers use the manifest to specify what external Intents (if any) should be delivered to the application's Components using Intent filters. Services, Activities and static broadcast receivers are declared in the manifest[51], while dynamic broadcast receiver is declared at runtime.

3.1.1 Exporting a Component

A Component that can only receive Intents from Components within the same application is considered private or unexported. If a Component can receive Intents from other applications it is considered to be public or exported. Exported Components expose their functionality to other applications, creating a potential security risk. But exporting is the foundation for the rich inter-application collaboration that encourages Component reuse.

Using the exported flag in the AndroidManifest.xml the developer can explicitly decide if the Component is exported or unexported. If this flag is omitted, it is up to the Android system to decide whether the component should be exported. The current criterion to export a component is, whether it contains at least one Intent filter. A Component exporting scheme is shown in Figure 4.

[49] Only exceptions are dynamic broadcast receivers that are declared at runtime and not in the manifest.

[50] Android (2013), http://developer.android.com/guide/topics/manifest/manifest-intro.html, 09-April-2013

[51] Chin et al. (2011)

A big security risk arises when a component gets exported by the Android system without the developer expecting it.[52] Exported Components also face the risk of receiving explicit Intents from other applications that will not match the Intent filters.

Unexported Components cannot receive explicit Intents from other applications (applications with a different UID[53]). Kantola et al. (2012) presented a heuristic to support the developer making the decision whether the component should be exported or unexported. The complexity of exporting Components is shown in Appendix 2.

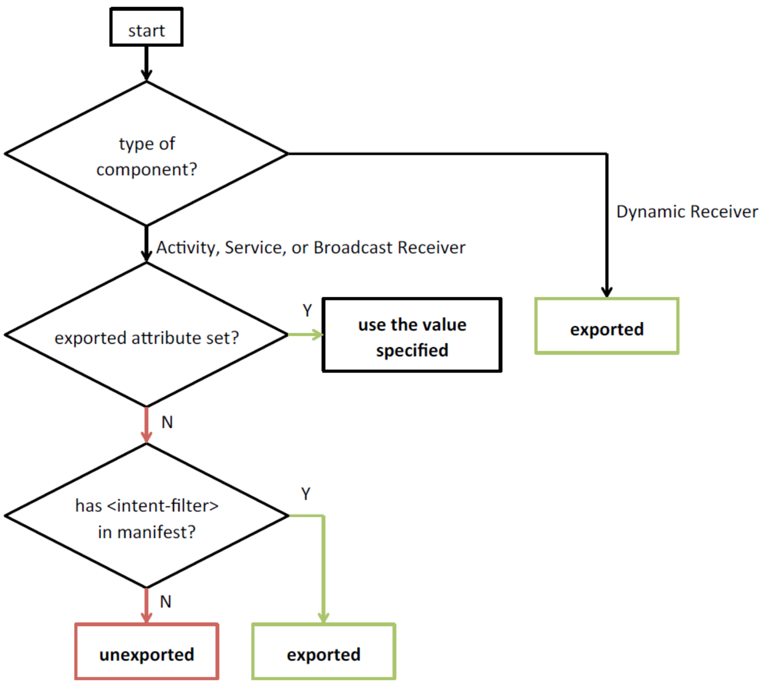


Figure 4: Component exporting scheme[54]

3.1.2 Android Permissions

Following the principle of least privilege, Android restricts access to the system resources. These restrictions are implemented in different forms. The sandbox and UID isolates the storage for each application. Some sensitive functionality of the system resources are protected by the lack of APIs to access them, while sensitive APIs that should only be used by trusted applications are protected through a security mechanism known as Permissions. There

[52] Kantola et al. (2012)

[53] Referring to the sandbox, every process has by default its distinct user ID with its own rights.

[54] Kantola et al. (2012)

is for example no API for directly manipulating the SIM card. Some sensitive/protected APIs are camera functionality, location data (GPS), phone functionality or messaging functionality (SMS/MMS), which are only accessible through the operating system.

An application has to define the needed permission in their manifest in order to gain access to protected API calls.[55] Applications can also use permissions to protect themselves. An application can specify that a caller (Intent sender) must have a specific permission to successfully send an Intent. This can be done by adding a permission requirement to a component's declaration in the manifest, setting a default permission requirement for the whole application, or adding permission checks throughout the code.

A broadcast Intent sender can limit the receivers of a broadcast Intent by requiring the recipient to hold a specific permission. This protection is only available to broadcast Intents and not to Activity or Service Intents.[56] Applications can use existing Android permissions or can declare new ones in their manifest. Permissions can be protected in security levels. There are four protection levels for permissions[57] shown in Table 3.

[55] Android (2013a), http://source.android.com/tech/security/index.html, 16-April-2013

[56] Chin et al. (2011)

[57] Android (2013), http://developer.android.com/guide/topics/manifest/permission-element.html, 16-April-2013

Normal	Permissions are granted automatically. Usually for minor consequences like vibrate. User can review but will not be explicitly asked or warned.
Dangerous	Permissions can be granted by the user during installation. If the permission request is denied, the application will not be installed. Used for dangerous rights that could reconfigure the device or incur tools.
Signature	Permissions are only granted if the requesting application is signed by the same developer that defined the permission. Signature permissions are useful for restricting component access to a small set of applications trusted and controlled by the developer. The unique developers signing key is used.
SignatureOrSystem	Permissions are granted if the application meets the Signature requirement or if the application is installed in the system application folder. Applications from the Android Market cannot be installed in the system application folder. System applications must be pre-installed by the device manufacturer or manually installed by an advanced user.

Table 3: Android manifest permission protection levels[58,59]

A Component protected with a Normal permission is essentially unprotected because any application can easily obtain the permission.[60] The Dangerous permission lets the user decide whether to grant the permission. When preparing to install an application the system displays a dialog, shown in Appendix 3, to the user that indicates the requested permissions and asks whether to continue the installation. By continuing the installation, the user has grants all of the requested permissions. The user has to accept all of the permissions as a whole. He is not able to grant or deny individual permissions.

Permissions are granted for as long as the application is installed, in order to support the seamless switching between applications.[61]

> *“Also, many user interface studies have shown that over-prompting the user causes the user to start saying "OK" to any dialog that is shown. One of Android’s security goals is to effectively convey important security information to the user, which cannot be done*

[58] Chin et al. (2011)

[59] Burns (2009)

[60] Chin et al. (2011)

[61] Android (2013a), http://source.android.com/tech/security/index.html, 16-April-2013

using dialogs that the user will be trained to ignore. By presenting the important information once, and only when it is important, the user is more likely to think about what they are agreeing to."[62]

Letting the user choose may be a risk in some cases, because not all users know a lot about Permissions. Since users cannot grant individual permissions they will accept all in order to install the application quickly. In these cases there is a possibility that a malicious application will be installed holding the Dangerous permission. This means developers can raise the security level by requiring Dangerous permission, but may still be attacked by malicious applications, that have been installed without the necessary awareness by the user.

Applications that use the hard to obtain Signature or SignatureOrSystem permission can be seen as private or unexported, hence they are quite secure (not considering attacks in which a developers signing key is compromised).[63]

3.2 Intent-based attack surfaces

The Intent based attack surfaces focuses on exported Components. "An attacker may be any untrusted application installed from an online application distribution platform as the Android Market, while a victim may be any application."[64]

In Section 3.2.1 it is shown how sending an Intent to the wrong application can leak user information. That data can be stolen by eavesdroppers and permissions can be accidentally transferred between applications. In Section 3.2.2 vulnerabilities related to receiving external Intents from other applications are shown. Accidently exported Components may be invoked by other applications in surprising ways or injected with malicious data.[65]

Exported Components with a hard to obtain Signature or SignatureOrSystem permission are not vulnerable to the presented attacks, because they cannot receive Intents from malicious applications. Unexported Components cannot receive Intents from any other applications at all.

[62] Android (2013), http://source.android.com/tech/security/index.html, 16-April-2013

[63] Kantola et al. (2012)

[64] Kantola et al. (2012)

[65] Chin et al. (2011)

3.2.1 Unauthorized Intent Receipt

Sending an implicit Intent does not guarantee that the Intent will be received by the intended recipient. Implicit Intents can be intercepted by malicious applications simply by declaring a matching Intent filter. If the Intent is not protected by a permission the malicious application lacks, it gains access to all of the data in the Intent.

Interception can also lead to denial of service or phishing, so called control-flow attacks.[66] Chin et al. (2011) described in detail the risks of a Broadcast Theft, Activity Hijacking, Service Hijacking, Intercepting Special Intents and why they are possible. In this thesis the associated risks of using a specific Component is more important than how the attacks are carried out. Kantola et al. (2012) summarized the risks for each component (except Content Providers[67]).

Broadcasts Intents are vulnerable to leak sensitive data, because passive eavesdropping is easily achievable. Eavesdropping means that a malicious application (Eve) defines a broadcast receiver with a matching Intent filters to receive the Intent. Eve just receives all broadcasts without other involved applications (Alice) noticing, shown in Figure 5.

Ordered broadcasts use a delivery chain with a priority rating. The highest priority receiver (Mallory) can modify (malicious data injection) or stop (denial of service attack) the propagation of the broadcast to the second highest receiver and so on. Ordered broadcasts are vulnerable to both active denial of service attacks and malicious data injections.[68]

[66] Chin et al. (2011)

[67] Content Providers cannot receive Intents; hence they are not directly vulnerable to attacks at the communication system.

[68] Kantola et al (2012)

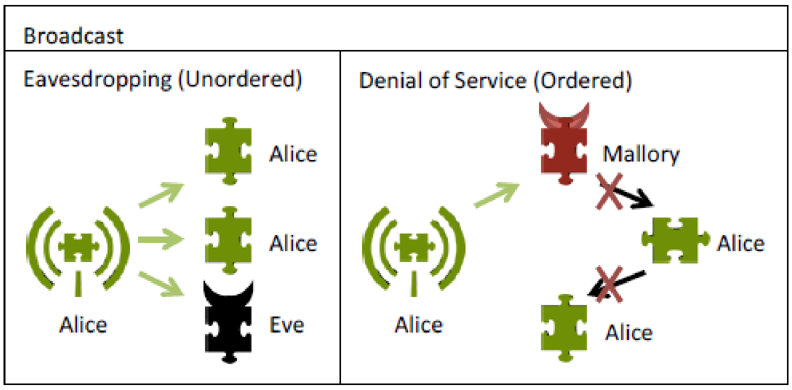


Figure 5: Broadcast eavesdropping and broadcast denial of service for Ordered broadcasts[69]

Sticky broadcasts have a higher risk of eavesdropping, because they stay available after they have been delivered and are re-broadcasted to future receivers.[70] Applications need a special permission[71] to send or remove a sticky Intent. This means that a sticky broadcast sender could be any application holding the STICKY–BORADCAST permission. But a receiver cannot force the sticky broadcast sender to hold a self-defined permission (e.g. Signature or Dangerous) in order to send a broadcast to the receiver.[72]

Activity and Service Intents can be intercepted and misused to start a malicious Activity or Service instead of the intended one. This is called a hijacking attack. It is a security issue, because an attacker can steal data from the Intent, hijack the user interface in a way that may be transparent to the user, and return malicious data to the sending component.[73] In Figure 6 is shown on the left how Alice unintentionally starts Mallory's Component instead of her own. On the right Mallory's Component returns a malicious result to Alice's Component. Alice still thinks the result's source is her own Component.

[69] Kantola et al. (2012)

[70] Chin et al. (2011)

[71] Value for the permission "android.permission.BROADCAST–STICKY"

[72] Burns (2009)

[73] Kantola et al. (2012)

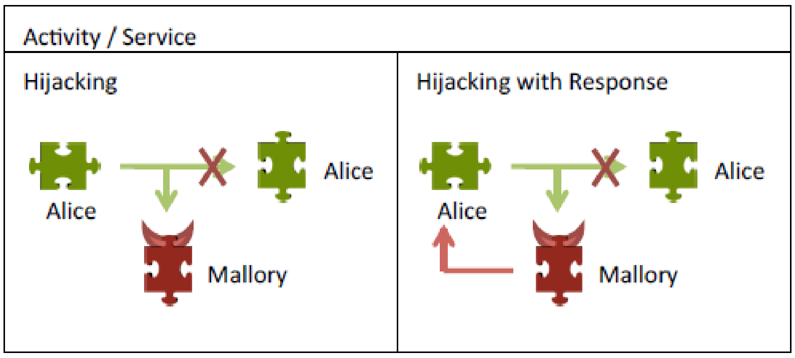


Figure 6: Activity and Service hijacking[74]

Permissions can be accidently transferred using special Intents. These can include URIs that reference data stored in application's Content Provider. The sender can set flags in the Intent to transmit the permission to read and write the stored data.[75] A compromised URI can lead to read and/or write access for a malicious application to all data stored by the Content Provider.[76]

Another special Intent is a pending Intent. It is created by an application and then passed to another in order to fulfill a described task with the permission and privileges of the creator, even if the creator application process was killed.[77] If a malicious application obtains a pending Intent (using one of the ways described above), the permissions of the Intent's creator can be assimilated.[78]

3.2.2 Intent Spoofing

"A malicious application can launch an Intent spoofing attack by sending an Intent to an exported Component that is not expecting Intents from that application."[79] If a victim executes an action, without checking the caller's identity, the attacker can trigger the Components action.

[74] Kantola et al. (2012)

[75] For example: "FLAG–GRANT–READ–URI–PERMISSION" or "FLAG–GRANT–WRITE–URI–PERMISSION"

[76] Chin et al. (2011)

[77] Android (2013), http://developer.android.com/reference/android/app/PendingIntent.html, 17-April-2013

[78] Chin et al. (2011)

[79] Chin et al. (2011)

Possible Intent spoofing attacks depend on exposed Components. Broadcast Receivers are vulnerable to broadcast injections, in which a receiving Component is tricked to believe that the incoming malicious Intent is sent from a trusted Component. If a malicious Intent is not recognized, it may take an inappropriate action (i.e. starting a Service or Activity) or the broadcast receiver operates on malicious Intent data.

System broadcast receivers are at danger, because they rely on actions that only the System can send.[80] If the receiver does not check the action of the Intent (thinking "only" Intents sent by the operating system can be received) an attacker may send an explicit Intent to the targeted receiver and trigger functionality that only the system should be able to do, because explicit Intents ignore Intent filters.[81]

An exported Activity or Service that is not secured with a strong permission may get started by a malicious application. A malicious Activity launch may cause annoyance to the user, because an unwanted interface could load. It could also affect the application state or modify data in the background. If the Activity uses the Intent's data without verifying its origin the application's data storage could be corrupted. The user could also be tricked by opening a "Settings" screen in the compromised application, but actually a "Settings" screen of a different application is loaded without the user noticing.[82]

3.3 Communication Security Conclusions

Developers have to be aware of the risks of intra- and inter-application communication. For intra-application communication explicit Intents or a Local Broadcast Receiver should be used.

The developer should also be cautious about exporting of Components unnecessarily. Exporting is a complex decision shown in Appendix 2. The export attribute in the manifest can be set by developers. If setting the export attribute to false no other application can send an implicit or explicit Intent to the Component. If a Component is exported the developer should make sure it can handle implicit and explicit Intents from malicious applications. Intent filters are not a security measure and can be bypassed with explicit intents.

[80] These Intents contain action strings that only operating system can add to a broadcast. For example "android.intent.action.BOOT–COMPLETED".

[81] Chin et al. (2011)

[82] Chin et al. (2011)

Receivers, especially System broadcast receivers, should always check the action-, data/type-, and category-field in order to make sure that an explicit Intent cannot trigger unwanted behavior. Before performing any operation, the Component should check the caller's identity, especially before sending out private data. Sticky broadcast should not contain sensitive data, because they cannot be secured with permissions like other broadcasts.[83]

If explicit Intents cannot be used, strong permissions to protect the Component should be specified. Results returned by other Components should be verified to ensure that they are from the expected source. Explicit Intents can contain null values for all other fields than the address of the Component. This may cause a null pointer exception, while comparing values. If a Component must handle inter- and intra-application communication, dividing it into separate Components should be considered.[84]

4 Solution Approach for myHealthAssistant

At the beginning the current- and desired state of myHealthAssistant is explained. Section 4.3 and the following show solution approaches and perquisites in order to reach the desired state.

4.1 Current State of MyHealthAssistant

In Seeger et al. (2012) requirements and corresponding decisions for myHealthAssistant such as using an event driven architecture and a smartphone as mediator are discussed.

Incoming sensor data is transformed into event types and published via an implicit broadcast. The action field of the broadcast Intent is used to define what information is sent. The action fields uses a channel name that is derived from the event type. Event types are defined hierarchically, starting from the ROOT_CHANNEL to the more detailed information separated by a dot, examples are shown in Table 4.

Weight Event	`ROOT_CHANNEL.Physical.Weight`
Gyroscope Event	`ROOT_CHANNEL.Physical.Inertial.Gyroscope`
Blood Pressure Event	`ROOT_CHANNEL.Physiological.Cardiovascular.BloodPressure`

Table 4: Event type examples

[83] Burns (2009), S. 16.

[84] Chin et al. (2011)

The most specific information is the last part of the String and the most general the ROOT_CHANNEL. If an application registers to the ROOT_CHANNEL, it receives all event types sent via the middleware. Any broadcast receiver can register to a channel and access the published events. A broadcast receiver can register for nodes and leaves in the event tree.

Every incoming event type that matches the conventions gets published on channels derived from its event type name. In Figure 7 the event types in Table 4 and their corresponding channels are shown.

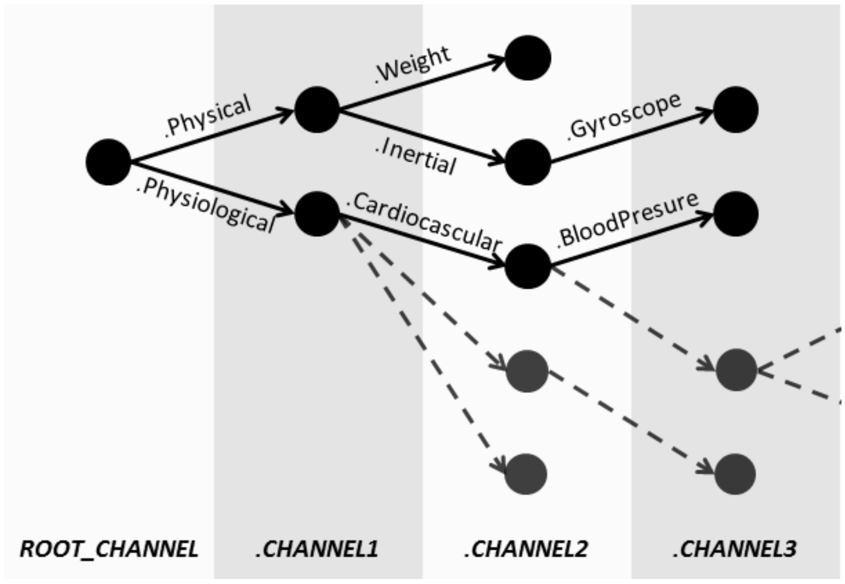


Figure 7: Derived channels from the event tree

An incoming weight event gets published via three implicit broadcasts, a blood pressure- and gyroscope event via four implicit broadcasts. All of them are published on the ROOT_CHANNEL. Third party applications can create new event types and publish them via myHealthAssistant which makes the event tree flexible and dynamic.

In the current state myHealthAssistant uses fast open communication channels, implemented as implicit broadcasts, but has the drawbacks of a non-restricted access to channels, no sensor managing functionality and it lacks the necessary information for an event type transformation.

4.2 Desired State of MyHealthAssistant

In order to get a better overview, the desired functionality is split into separate modules as shown in Figure 8.

The Message Handler can talk to parts of myHealthAssistant via interfaces and applications by sending messages via the application-specific communication channels. The interfaces to the Transformation Manager, Security Manager and Sensor Modules have to be implemented, as well as the application-specific communication channels.

Security Manager decides if an application has the right to access (subscribe) a channel and may change the access rights at runtime. The Transformation Manager has the knowledge how to transform currently available event types into new ones. New transformation rules can be downloaded from an online repository at runtime.

The decision to start, stop or change the configuration of a sensor is made by the Message Handler, based on the knowledge about interested applications at sensor readings (event types) and possible configurations of each sensor. The Message Handler decides and sends a corresponding message to a Sensor Module or a third party application that provides events (publisher). The Sensor Module and third party application have to implement the functionality in order to react accordingly.

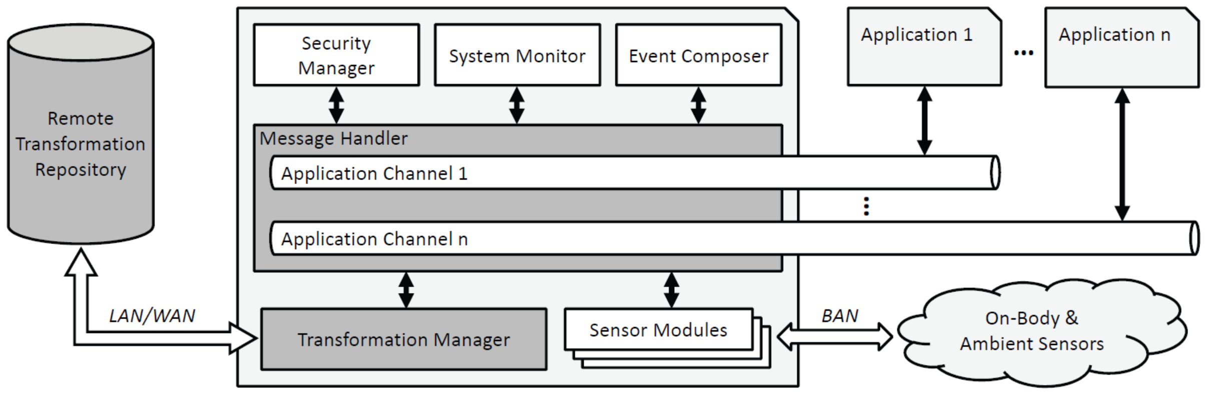


Figure 8: MyHealthAssistant desired state[85]

[85] Seeger et al. (2013)

The System Monitor and Event Composer are already implemented. The System Monitor is responsible for monitoring the overall system status. It reacts accordingly to a low battery status for example.

The Event Composer identifies general situations on which the system has to react to (e.g. critical vital signs), based on sensor and application related events. In reaction it creates corresponding derived events and sends a message (E-Mail, SMS, etc.) to predefined recipients.[86]

4.3 Integrating Publish/Subscribe

Event based systems provide a loosely-coupled communication and therefore a very flexible multi-sensor and multi-actor communication.[87] MyHealthAssistant needs to support changing sensors and running applications over time as well as handling sensors that send their readings in an event-driven manner. Those characteristics are supported by the loosely-coupled publish/subscribe communication scheme.[88]

The introduction of a Message Handler using the publish/subscribe scheme is the foundation to provide the desired functionality. In order to provide publish/subscribe the following new event types are introduced: Subscribe, Unsubscribe, Advertise and Unadvertise are used. The events are explained in Table 5.

Subscribe	Applications register their interest in an event type/channel.
Unsubscribe	Applications unregister their interest in an event type/channel.
Advertise	Sensor Modules or applications notify the Message Handler that they (can) provide a specific event type.
Unadvertise	Sensor Modules or application notify the Message Handler that they do not provide a specific event type anymore.

Table 5: New event types for publish/subscribe

[86] Seeger et al. (2013)

[87] Seeger et al. (2012)

[88] Eugster et al. (2003)

The two most widely used variants of publish/subscribe are topic-based and content-based, will be considered for our solution approach.[89] Topic-based publish/subscribe uses keywords to subscribe for events. It is implemented in the current approach in myHealthAssistant by subscribing for a channel and receiving all published event types on this channel. The subscriber has to filter every incoming event and decide if the carried information is useful or not.

Content-based publish/subscribe provides a more complex filtering. An application subscribes to an event with an event specific value range. For example a content-based subscription would be to receive heart rate events only if the heart rate is higher than 160 bpm. Using this approach the filtering effort is done by the middleware and the application only receives heart rate events that are higher than 160 bpm.

Using the topic-based approach the application receives all heart rate events and has to check every incoming heart rate event if it is higher than 160 bpm. This will result in more filtering effort for the application. In Table 6 the main differences are shown.[90]

	Pros	Cons
Topic-based	• Easy to implement • Fast event routing	• Subscribers have to check every event for a desired value
Content-based	• Subscribers only receive desired values of an event	• More complex • Slow event routing • Message Handler has to check every incoming event and confront subscriber's filters

Table 6: Pros and cons of Topic-based and Content-based scheme

In some cases content-based variant excels in others topic-based is more useful. Approximating the calculation effort for each solution may help with the decision.

For example, 5 applications are interested in only a small range of heart rate values and 100 heart rate events are sent to the middleware. Using a content-bases variant the middleware has to check 100 events and confront the subscriber's filters and only sent matching events to the subscribers. As a result, a lower amount of events is sent to the subscribers and the

[89] Eugster et al. (2003)

[90] Eugster et al. (2003)

subscribers do not have to filter events. The Topic-based variant would require each subscriber to check 100 events. As a result every heart rate event is sent to each application, all applications have to check incoming events, resulting in 500 events checked in total.

On the other hand, there are sensors where it is not useful to specify small ranges, like accelerometer or ECG events. There might also be developers that do not know or trust the specified filters and subscribe without a specific range. In this case, the Message Handler and the subscriber will check the event which increases the overall calculation effort. If applications decide to provide new events (not known by myHealthAssistant) an event filtering in the middleware is more complex.

Considering pros and cons as well as the explained example, the fast routing and easy to implement topic-based scheme is chosen. The burden of filtering events for their specific values remains on the application side. The Message Handler tries to reduce the amount of events handled by stopping unnecessary sensors or by reducing their transmitting frequency.

4.4 Identifier for Applications and Restricted Access to Channels

The basic prerequisite for publish/subscribe is the ability to (uniquely) identify subscribers and publishers. The identification of a subscriber in Android using Intents is a problem, because the system does not provide a function to verify the Intent-sender. A malicious application can send subscriptions and pretend to be different application. In this case, the Message Handler cannot identify the malicious Intent.

To verify an Intent sender, a unique identifier for every application has to be assigned and included in every message sent to the Message Handler. With respect to security it should not be possible for malicious applications to fake the unique identifier and retrieve sensitive information. In Section 4.5 it is shown that sensitive information will not be leaked, even if the unique identifier is faked.

In the Android system, the package name specified in the AndroidManifest.xml is unique. It is not possible to install two applications with the same package name. It is also impossible to publish an application to the Google Play store if an application with the same package name already exists.

*“**Caution:** Once you publish your application, you **cannot change the package name**. The package name defines your application’s identity, so if you change it, then it is considered to be a different application and users of the previous version cannot update to the new version.”*[91]

The package name is a full Java-language-style package name for the application. It is declared in the AndroidManifest.xml and can be chosen by the developer, as long as it matches the conventions.[92] For example, an application with the package name “com.walmart.shopping” does not necessarily belong to Walmart and/or the owner of the domain. Hence the publisher of an application cannot be inferred from the package name, but the package name is still unique.

MyHealthAssistant identifies subscribers and publishers using the package name. The Security Manager provides the functionality to verify if the package name is valid and if the application is installed on the Android device. The Message Handler sends a request containing event type and package name to the Security Manger, which grants or rejects the subscription. If access rights change at runtime, the Security Manager reviews all current subscriptions with the new access rights and unsubscribes applications if necessary. If a subscription is invalidated a notification is sent to the affected application.

4.5 Secure Communication Approach

The authors of Chin et al. (2011) and Kantola et al. (2012) have shown vulnerability in the Android communication system. In addition they identified the unclear distinction between intra- and inter-application communication system as the main problem. By searching and explaining the vulnerabilities in the communication they also mentioned secure ways to use the inter-application communication, as long as the operating system is not compromised.

By using explicit Intents malicious applications cannot intercept the message and, thus, no sensitive data can be leaked. Securing the communication by using Android specific communication procedures will save the costs for encrypting the communication between

[91] Android (2013), http://developer.android.com/guide/topics/manifest/manifest-element.html#package, 22-April-2013

[92] Android (2013), http://developer.android.com/guide/topics/manifest/manifest-element.html#package, 22-April-2013

applications and the Message Handler. In Section 6 Evaluation the performance of the application-specific communication channels is compared to the open communication channels. A solution that uses encryption for restricting access will have to compete with the provided benchmark results in this thesis.

Therefore, explicit Intents will be used for our solution approach. The transmitted package name is used to address explicitly the application/Component. The package name in the subscription message can be manipulated by a malicious application, but eavesdropping and leaking of sensitive data to unauthorized applications is not possible due to the Security Manager that restricts the access to events/channels by using the package name.

A malicious application can subscribe to an event with a false package name, but the package name (application) has to have the right to receive the event in order to pass the Security Manager check. If it passes the security check, the explicit addressing makes it impossible for the malicious application to receive the events. This way sensitive data will not be lost, but the internal list of publisher and subscribers of myHealthAssistant can be compromised by malicious applications.

To reduce the risk of a compromised list of publishers and subscribers a Dangerous permission can be introduced. Also the Security Manager could implement methods to verify that a package name is from a valid application. Hopefully, a later version of Android will implement a method to reliably verify an Intent sender.

There might be a way to secure the communication using certificates and the Advanced Android Definition Language (AIDL) but it would probably raise the barrier for other developers to use myHealthAssistant. A certificate has to be requested, created and sent to the developer before an application can subscribe for event types. Using certificates would make it harder to change access rights at runtime. If the overhead for managing certificates is too big, it will annoy developers. Using AIDL can lead to an advanced solution, but it is also more complex. This thesis does not look at a solution using AIDL. However, it will provide benchmark results that future AIDL solutions will have to compete with.

4.5.1 Explicit Solution

The first approach used explicit Intents to address every Component utilizing the corresponding *ComponentName*. In that solution a subscription contains the *ComponentName* which consist of the package name and the class name inside the Component. In this case, the access rights in the Security Manger can be set very granular to every broadcast receiver/Component that tries to subscribe for an event as shown in Figure 9. Every broadcast receiver within an application has to register a broadcast receiver and subscribe for event types.

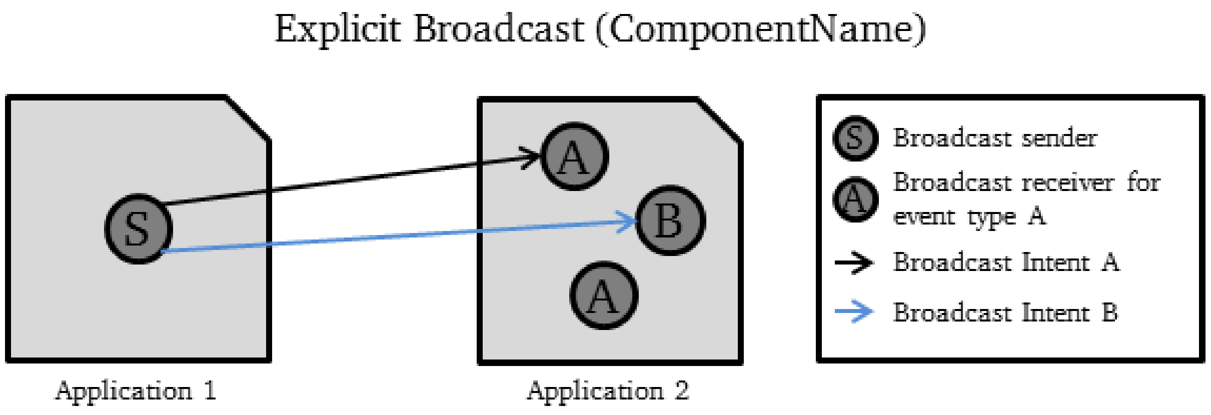


Figure 9: Explicit broadcast using *ComponentName*

During implementation a problem with explicitly addressing a receiver appeared. Dynamic broadcast receivers cannot be addressed by explicit Intents.[93] In that solution, a developer is limited in using only static broadcast receivers. Static receivers can only call static methods of other classes and have a short lifecycle. A new instance of the static receiver is created for every incoming broadcast and disappears after the code is finished.

Benchmarks of a prototype using this solution showed a bad performance in the CPU utilization and delivery ratio, shown in 6.2 Measurement Analysis.

This approach is discarded. It satisfies the requirements for myHealthAssistant, but it would limit developers to use only static broadcast receivers with a much worse performance compared to the original Open channel implementation.

[93] Kantola et al. (2012)

4.5.2 Explicit Combined with Implicit Solutions

On lesson learned from the explicit Intents using the *ComponentName*, a solution which allows developers to use dynamic and static broadcast receiver is needed. Research lead to the *setPackage()* method where Intents are delivered explicitly to the application's boundaries, but are implicitly resolved within the application. This allows developers to use dynamic broadcast receivers, which are bound to the Component lifecycle that registered it.

The new subscription contains the package name and the event type. That is the main difference to the explicit approach.

The Security Manager now handles subscriptions for each application and not for each broadcast receiver. Using *setPackage()* applications subscribe for channels instead of the Explicit solution where each broadcast receiver had to subscribe for a channel. The difference is shown in Figure 9 (*ComponentName*) and Figure 10 (*setPackage*).

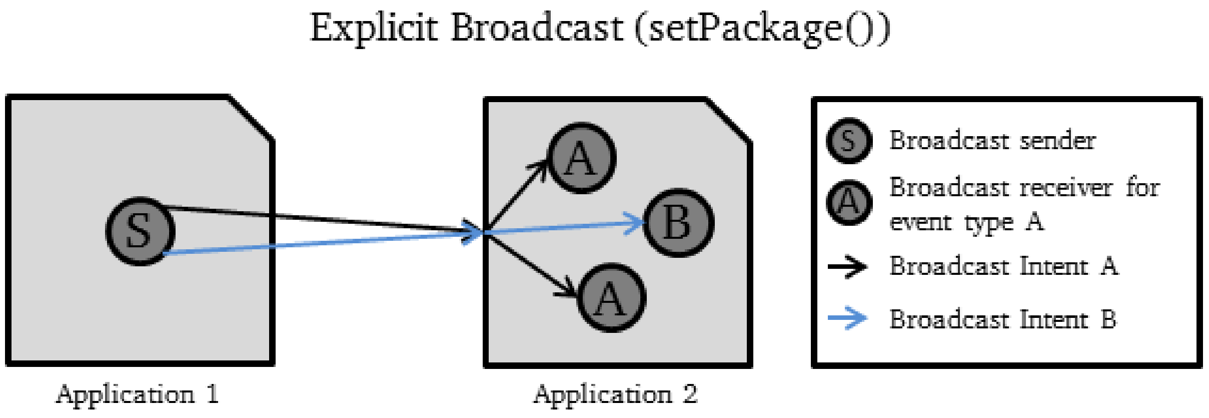


Figure 10: Explicit broadcast using *setPackage()*

According to the paper Kantola et al. (2012), there is a bug in the Android source code that does not limit the recipient to specific applications for broadcast Intents.[94] This bug has been fixed with Android version 4.0 Ice Cream Sandwich.

> *"…Starting with ICE–CREAM–SANDWICH, you can also safely restrict the broadcast to a single application with Intent.setPackage()."*[95]

[94] Kantola et al. (2012)

[95] Android (2013), http://developer.android.com/reference/android/content/BroadcastReceiver.html, 28-March-2013

A perquisite of Android version 4.0 or higher will reduce the amount of compatible smartphones from 94.5 % to 55.9 % off all current Android phones, shown in Appendix 4. In this case, the tradeoff between more comfort for developers and a better performance compared to a smaller range of matching Android smartphones has to be made.

In order to support other developers the cut in range is acceptable compared to the gain in comfort and performance (compared to the Explicit solution). The setPackage solution is chosen and implemented.

5 Implementation

This Chapter explains how the publish/subscribe scheme, start and stop logic for publishers as well as Transformation- and Security Manager interface are implemented. In addition problems that occurred are explained.

The source code for a sample application that subscribes for a heart rate event and that can send an acceleration event to myHealthAssistant is shown in Appendix 5. In order to use myHealthAssistant the sample application has to import the myHealthAssistant.jar file.

5.1 Implementing Publish/Subscribe

In order to implement the new features the existing Service of the Message Handler in myHealthAssistant is extended. The current Service gets started automatically and a regular heart beat checks and restarts the Service if necessary.

Table 7 shows how an Event and the inherited *Un/Subscribe* and *Un/Advertise* management events are implemented. Management events expand the Event by the necessary information of the *packageName* and affected event type. The String *JSONEncodedProperties* in the Advertise event can contain extra information about different configurations of the sensor.

Event	`Event(String eventType, String eventID, String timestamp, String producerID)`
Subscribe	`Subscribe((Event), String packageName, String subscriptionSensorReadings)`
Unsubscribe	`Subscribe((Event), String packageName, String unSubscriptionSensorReadings)`
Advertise	`Advertise((Event), String packageName, String advertisedEventType, String JSONEncodedProperties)`
Unadvertise	`Unadvertise((Event), String packageName, String unadvertisedEventType)`

Table 7: Event structure for publish/subscribe scheme

The data transmitted with management events is stored in the *subscriberChannelHashMap* and the *advertisementHashMap* shown in Table 8.

Subscriptions	`subscriberChannelHashMap<String, List<String>>`
Advertisements	`advertisementHashMap<String, List<ProducerDetails>>`

Table 8: *SubscriberChannel*- and *advertismentHashMap*

The *subscriberChannelHashMap* uses the subscribed event type as a the key and a list of all package names containing the subscribers as a value in order to explicitly address the subscribers. In the *advertisementHashMap*, the event type is also used as key, but the value is a list of constructs, containing the *producerID* and the *packageName*. The additional *producerID* is used to support multiple sensor readings from one publisher. If there are three accelerometers from the same publisher the middleware could not distinguish them without the *producerID*.

Using HashMaps a data structure supports a quick access (O(1)) to the list of producers or publishers for a specific event type, even with a lot of different event types.

5.2 Data consistency on crashes

A *HashMap* is not a persistent data storage. If the Service of myHealthAssistant stops due to an error, the *advertisementHashMap* and the *subscriberChannelHashMap* with all entries are lost. In order to store the data, the last valid un/subscription for each application and event type is saved in a SQLite database. When the Service restarts after a crash, the *subscriberChannelHashMap* data gets restored from that database. A database for the *advertismentHashMap* can be implemented as well.

Even writing every valid management event received by the middleware in a SQLite database will not influence the overall performance, because management events do not occur often. The overall recovery process from the database into the HashMaps has to be defined in detail when the Transformation Manager is implemented. The implementation of the Transformation Manager matters, because the restore time affects the content of messages sent to subscribers, publishers and the Transformation Manager about available, requested and transformed event types.

5.3 Start and Stop logic for Publishers

The Message Handler decides if sensors should be started or stopped, based on information about current subscribed applications and available sensors.

A Start event is sent to a publisher, when the first application subscribes for the event type. A Stop event is sent when the last application unsubscribes from the event type.

Using a tree structure with the ability to subscribe to leafs and nodes makes this process more complex. An un/subscription to the ROOT_CHANNEL for example affects every available publisher in the event tree.

A subscription to a node affects all children and the corresponding leaves as shown in Figure 11. For leaf A the blue line shows all the channels that have to be checked for subscribers in order to decide if leaf A should be started or stopped. A publisher can also be a node as shown for node B. Applications can create their own event types and publish them using myHealthAssistant.

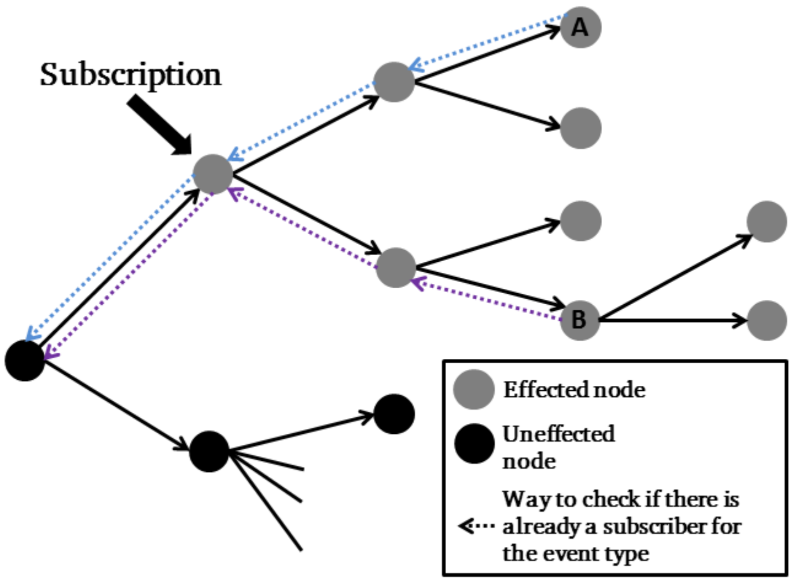

Subscription
A
B
Effected node
Uneffected node
Way to check if there is already a subscriber for the event type

Figure 11: Subscription to a node and the effects on the event tree

The following steps are taken in order to send a Start or Stop event to all publishers.

Subscription:

1. Get all matching publishers = all advertised publishers that start with the subscripted event type/channel
2. Check for each matching publisher
   2.1. if there is no subscriber, send a Start event to the publisher
3. Add the subscriber to the *subscriberChannelHashMap*

Unsubscription:

1. Get all matching publishers
2. Check for each matching publisher
   2.1. if there is exactly one subscriber, send a Stop event to the publisher
3. Delete the entry in the *subscriberChannelHashMap*

Using a tree structure with subscriptions to leaves and nodes, this implementation is expensive and should be optimized. *Subscribe* and *Unsubscribe* events do not occur often but there is a lot of optimization potential.

5.4 Security- and Transformation Manager

The Security Manager combines the knowledge about channels and applications that are allowed to subscribe to them. The Message Handler can use the method `hasPermission(String packageName, String channel)` to verify if an application has the rights to subscribe to a channel or not. In order to support changing access rights at runtime, the Security Manager uses the method `checkPermissionsForEventType(String eventType, List<String> allowedSubscribersPackageNamesList)` to check if a current subscriber has lost the right to receive the event type. If a subscriber lost the access right an unsubscription message is sent by the system, which deletes the *packageName* of the subscriber in the *subscriberChannelHashMap*. The affected application is notified via an announcement.

The Transformation Manager and the Message Handler communicate via events. The *EventTransformationRequest* and the *EventTransformationResponse* are shown in Table 9. The *requestType* specifies the kind of request, for example to request or stop a transformation. A request for a transformation requires all advertised events and the targeted event type. The Transformation Manager subscribes to advertised events that are needed for the transformation and sends an advertisement to the Message Handler.

`Event(String eventType, String eventID, String timestamp,`
`String producerID)`

`EventTransformationRequest((Event), int requestType,`
`String[] advertisedEvents, String eventSubscription)`

`EventTransformationResponse((Event), String requestEventID,`
`boolean success)`

Table 9: *EventTransformationRequest* and *EventTransformationResponse*

5.5 Intent Delivery

The method *setPackage()* is used to deliver messages between the Message Handler and applications. Applications know the package name of myHealthAssistant and can send management- and self-produced events. MyHealthAssistant publishes incoming events to subscribers of the corresponding channels.

For each channel the *subscriberChannelHashMap* has a list of package names. For each entry in the package names list a broadcast is sent using the *setPackage()* method. The Message Handler and applications use the *setPackage()* method to send Intents. This creates the application-specific communication channels. Other applications cannot access these channels.[96]

If there is no entry for a channel in the *subscriberChannelHashMap*, than there is no subscriber and the event is not published in this channel.

A weight event for example requests the list of subscribers at the *subscriberChannelHashMap* for three channels:

- ROOT_CHANNEL.Physical.Weight
- ROOT_CHANNEL.Physical
- ROOT_CHANNEL

Malicious applications can fake the *packageName* String in a Management event and trick the Message Handler and Security Manger. But no sensitive data will leak, because events get only transmitted to valid application's package names, using the *setPackage()* method. A malicious application cannot get sensitive data, but it can compromise the *subscribersChannelHashMap* and *advertismentHashMap* in this implementation.

Management events could be secured by a Dangerous permission, encryption or maybe the Android system will provide a method to reliably check the sender of an Intent in later versions.

5.6 Announcements

The Message Handler informs applications about events that happen. For example it responds to applications that the subscription was successful, that the subscriber does not have permission to access the channel or that it is already subscribed. Applications can use Announcements to request a list of all available event types from myHealthAssistant.

An `Announcement((Event), String transmittedEventType, String packageName, int announcement)` has to transmit the event type, package name and the type of announcement. The package name is required to respond individually to an announcement sender. Using the *setPackage()* method, announcements are only transmitted to affected applications over the application specific channel. The most commonly used announcements are shown in Table 10.

[96] Chin et al. (2011) Not considering attacks at the Android system.

Message Handler broadcast to all applications
ADVERTISMENT_NEW_EVENT_TYPE_AVAILABLE
UNADVERTISMENT_EVENT_TYPE_NOT_LONGER_AVAILABLE
Message Handler response to application used *setPackage()*
ADVERTISMENT_SUCCESSFULL
UNADVERTISMENT_SUCCESSFULL
SUBSCRIPTION_SUCCESSFULL
SUBSCRIPTION_SUBSCRIBER_ALREADY_IN_LIST
SECURITY_PERMISSION_REMOVED
EVENT_TYPE_DOES_NOT_MATCH_NAME_CONVENTION
INVALID_EVENT_TYPE_ARGUMENTS
ALL_AVAILABLE_EVENT_TYPES
Applications to Message Handler
GET_ALL_AVIALABLE_EVENT_TYPES

Table 10: Common Announcements

6 Evaluation

The Explicit and setPackage solution fulfill the specified requirements. But the main task of myHealthAssistant is to publish incoming events to interested applications. The performance in publishing events in each solution has to be compared, in order to decide if the gain in functionality is worth the loss in performance.

6.1 Experimental Setup

The main task of myHealthAssistant is to publish all incoming events to the matching subscribers. For this purpose the system's behavior on an increasing amount of events per second as well as an increasing amount of subscribers is analyzed.

Event generators, which run in the same process as the middleware, are used as operating sensor modules. Each event generator creates a heart rate event with 1 Hz frequency and sends it to the Message Handler. The Message Handler transmits it to an application with up to 11 broadcast receivers, starting with one and increasing it by two. The broadcast receivers are all subscribed to heart rate events and the first receiver counts all incoming events. All test results presented are the average of three test runs over 100 seconds. The test starts with 5 heart rate events per second and increases up to 20 events per second, in steps of 5.

For the analysis, a HTC One V smartphone with a 1 GHz Tegra 2 single-core processor, 512 MB memory and Android 4.0.3 is used. The original-, the Explicit- and setPackage solution are measured. The original implementation represents open channels. The Explicit- and setPackage implementaiton uses application-specific communication channels.

The experimental setup was designed to compare the implicit and explicit solution. There is a problem while evaluating the setPackage solution. Evaluating setPackage with one application containing 11 receivers would lead to wrong data. In this special case setPackage behaves like the Open channels implementation. The difference between Open channels and setPackage() will only show by using different applications containing one receiver each.

Open channels and the Explicit solution does not show a significant difference between one application and 11 receivers compared to 11 applications with one receiver. Whereas *setPackage()* shows a big difference due to the mixing between explicit and implicit Intent resolution, shown in Figure 12. In order to get a comparable result *setPackage()* is evaluated with 11 different applications each containing one receiver. Only the first application's receiver will count the received heart rate events.

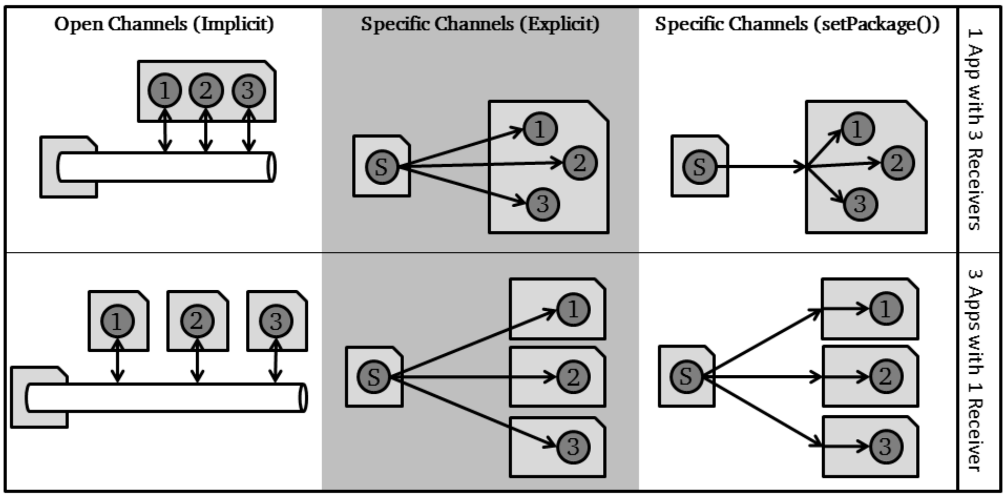


Figure 12: Difference by evaluating Open vs. Specific channels

A problem occurred, because the HTC one V can only run 10 Activities at the same time. The test with 9 applications and myHealthAssistant is possible, but 10 applications and myHealthAssistant does not work. Starting Activity number 11 stops the first Activity. Therefore, there is no data about 11 subscribers for the setPackage solution.

6.2 Measurement Analysis

The measurements of Open channels use blue, Explicit solution green and setPackage solution orange colors. The evaluation starts with the CPU utilization (of the middleware), delivery ratio and ends with the memory usage. In the diagrams each block of three measurements shows the measurements of the: Open channels on the left side, Explicit solutions in the center and setPackage on the right side.

All three solutions are compared with each other, shown in Figure 13, 15, 17 and 18. These figures most important information is that the Explicit solution has a worse performance than the other solutions. In order to show the difference between Open channel and the setPackage solution in more detail, extra diagrams comparing these two are included. Figure 14, 16 and 17 (on the right) compare the Open channel with the setPackage solution.

The first important criterion is the CPU utilization. The explicit solution uses a lot more CPU cycles than the other two implementations. The difference is small at 3 subscribers and 15/20 events per second (3 %/5 %) but the utilization increases very fast. For 5 subscribers it is already at 6 % for 15 events/s and at 12.7 % for 20 events/s which is already 3 times higher than the setPackage and Open channels solution. With an increasing amount of subscribers the utilization increases to 76.6 % (9 receivers and 20 events/s) which is more than 15 times higher than the setPackage solution (4.8 %). As result the Explicit solution cannot compete with the Open channel and setPackage solution.

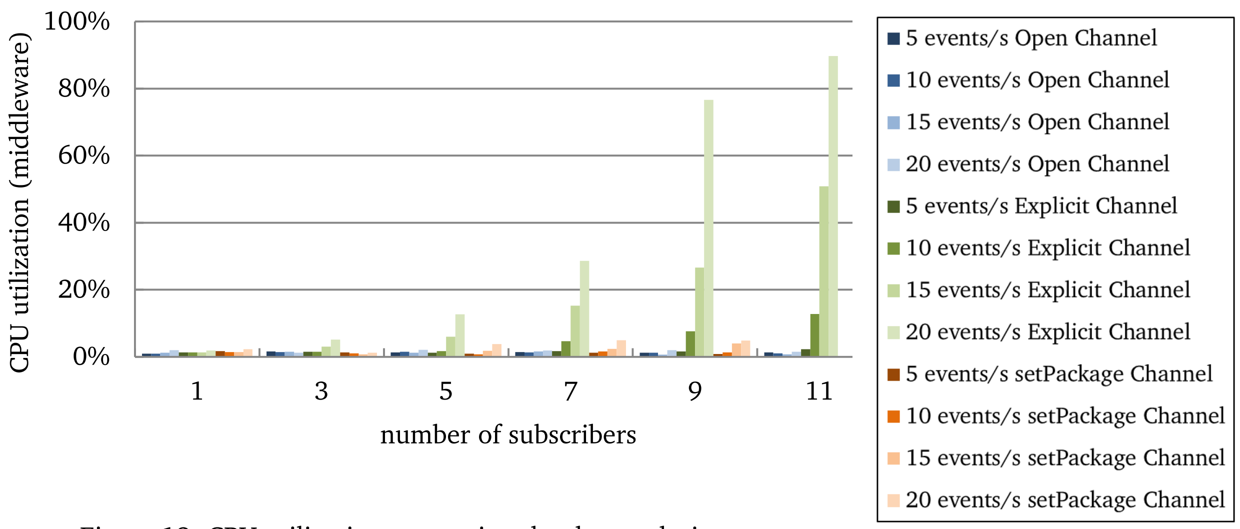


Figure 13: CPU utilization comparing the three solutions

Comparing Open channels with setPackage is more interesting as shown in Figure 14. They are pretty equal until 5 subscribers while handling less than 20 events/s. Handling 20 events/s, the CPU utilization for setPackage is 3.8 % compared to the 2.1 % for the Open channels. With an increasing amount of subscribers and events/s setPackage uses more than twice the CPU utilization, but stays under 5 %. When Handling 9 subscribers and 25 events/s the utilization went up to 8.2 %.

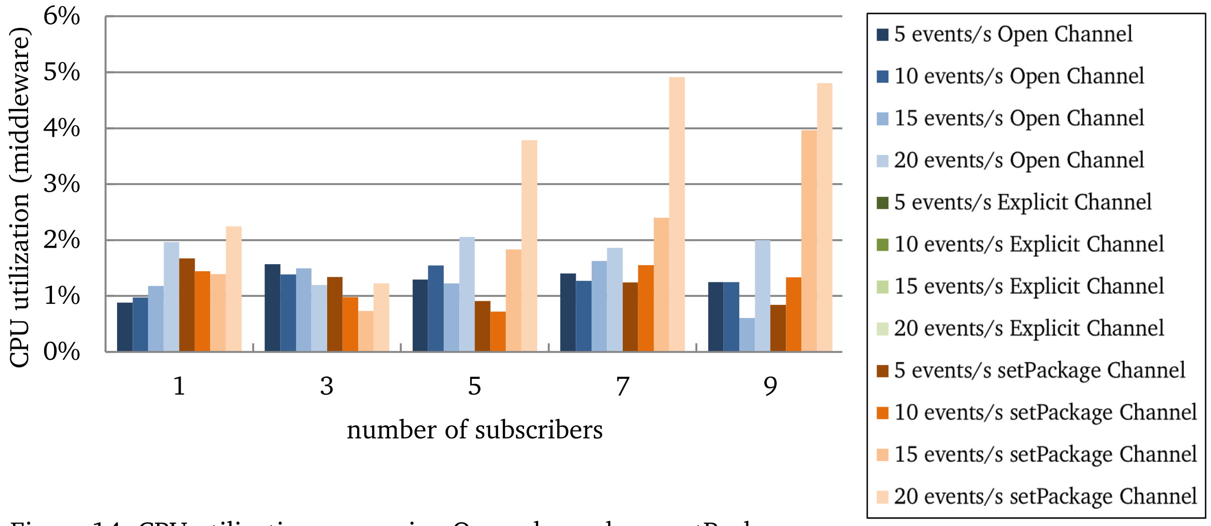


Figure 14: CPU utilization comparing Open channels vs. setPackage

The delivery ratio shows that the Explicit solution starts to lose events already with three subscribers and 15 events/s or more. The delivery ration is lower than 91 % while handling 9 subscribers and 15 events/s or more, shown in Figure 15.

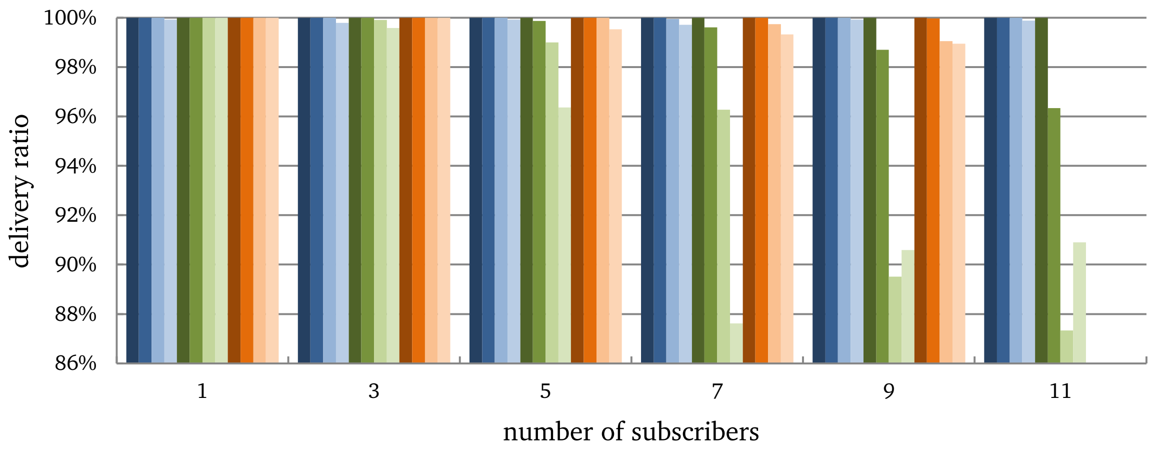


Figure 15: Delivery ratio for the three solutions

SetPackage starts losing events with 5 subscribers and handling more than 20 events/s, but the rate is still high with 99.53 %. Handling more than 7 subscribers the rate drops to 99.32 %, while handling 15 events/s. Open channels have problems handling 20 events/s independent from the number of receivers. The mean value is 99.85 % events received. SetPackage delivery ratio is better than Open channels while handling 3 subscribers and even with 5 subscribers, but it gets worse with more than 5 subscribers, shown in Figure 16, 17 (on the right side).

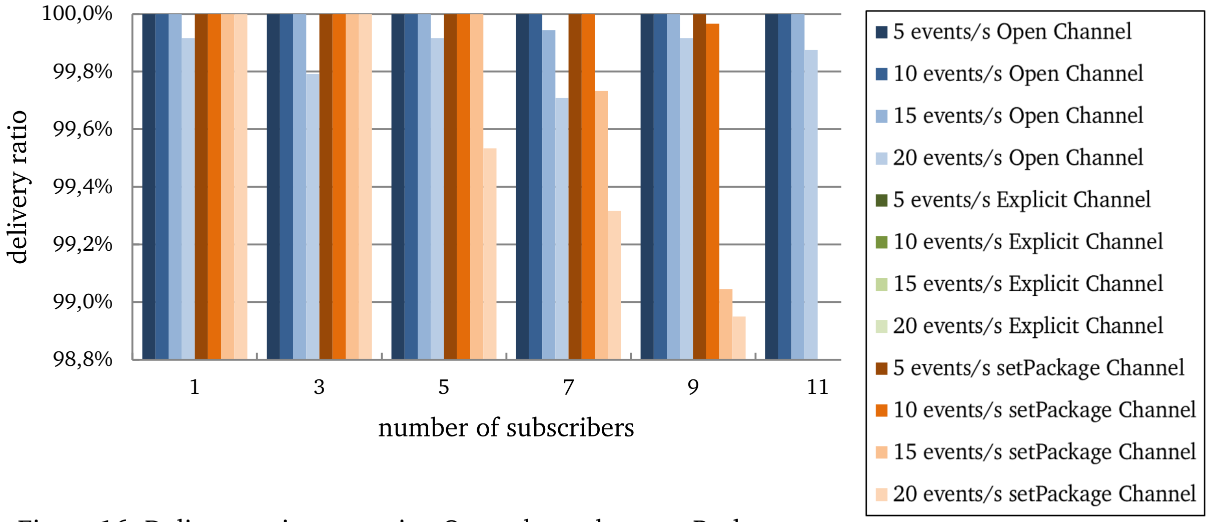


Figure 16: Delivery ratio comparing Open channels vs. setPackage

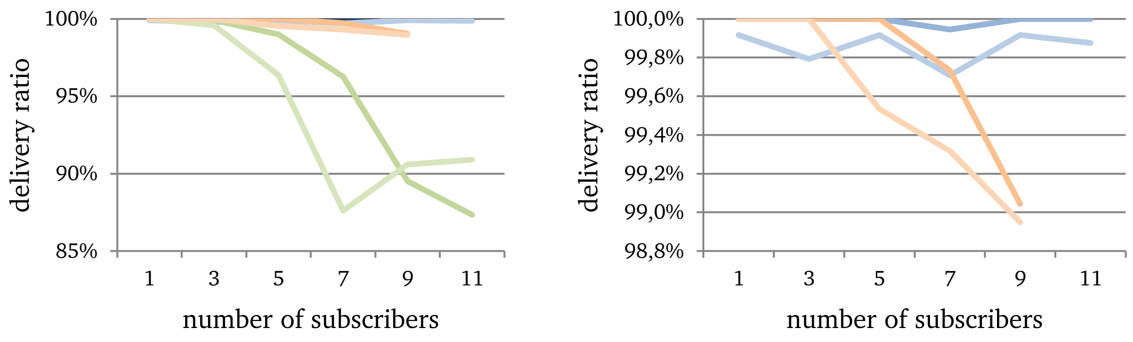


Figure 17: Delivery ratio for three solutions (left) and Open channels vs. setPackage (right)

There are no big differences in the memory usage. The usage increases independently from the solution with a growing workload, shown in Figure 18. SetPackage has the highest value, but it also has the most features implemented. A current low-end smartphone possesses at least 512 MB main memory and setPackage would use in the worst case only around 1.6 % of that.

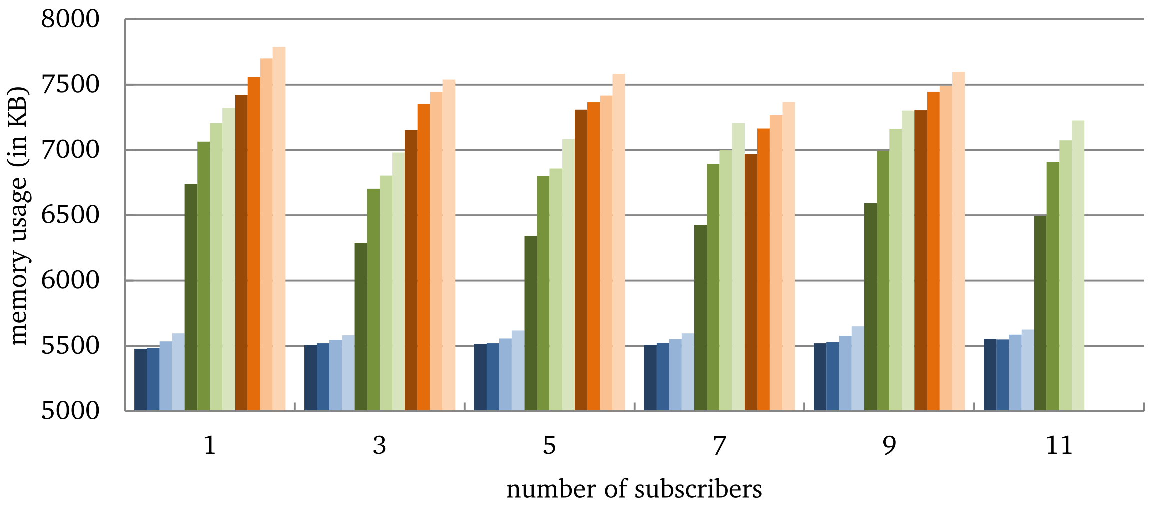


Figure 18: Memory usage for the three solutions

The evaluation shows that setPackage is much more efficient than the Explicit solution. Even compared to Open channels the performance is equal for 5 or less subscribers and handling up to 20 events/s. The gain in functionality and a perfect delivery ratio for up to 3 subscribers shows that setPackage is a feasible and efficient solution.

7 Conclusions

The main task of this thesis was to design and implement a solution that satisfies the need for restricted access to event types/channels, more sensor management functionality and information needed for supporting event transformations.

In order to provide a restricted access to channels a publish/subscribe scheme is implemented. The package name of an application enables the middleware to identify publishers and subscribers. Based on the knowledge about subscribers and publishers, myHealthAssistant can decide whether to start or stop a sensor and it provides the necessary data to request a transformation of an event type. The mixture between explicit and implicit Intent resolution, using *setPackage()*, defines the application-specific communication channels from the boundaries of the middleware to the boundaries of each application. Within these boundaries implicit Intent resolution is used, providing a maximum of flexibility to application developers.

Comparing the small loss in performance to the gain in functionality and security, the introduction of application-specific communication channels are a successful addition to myHealthAssistant. The first solution approach was inefficient and uncomfortable for developers. The second approach improved the efficiency and does not limit developers to use only static broadcast receivers. MyHealthAssistant supports Android version 2.3 or higher, but due to a bug in the Android source code, the full potential to restrict malicious applications to access sensitive data is achieved by using Android version 4.0 or higher.

The evaluation was not designed to measure the effects of started and stopped sensors, because the necessary implementation in the sensor modules is not available yet. The performance and energy saving potential of a reduced amount of events handled by the system, due to no unnecessary running sensors, is not considered yet. Another addition to energy saving is the possibility to reduce the amount of sensors using event transformation. For example an ECG stream can be transformed into heart rate events. This makes a heart rate sensor unnecessary while using an ECG sensor as discussed in Seeger et al. (2013).

8 Future Work

Security: To increase the security and robustness against malicious applications management events should be secured with a Dangerous permission, an encryption or other features. Otherwise malicious applications can compromise the *subscriberChannelHashMap's* and *advertisementHashMap's* data.

Robustness: Every application can produce own event types and use myHealthAssistant to publish it to all subscribers. The only restriction is to check if the event type starts with the ROOT_CHANNEL. But there could be null values in other fields of an event. That could crash subscribers. A Dangerous permission to send events to myHealthAssistant could increase the robustness. A restricted access and checking all incoming events can cause a drop in the performance, due to the high amount of events sent by the Sensor Modules. A better and more secure way would be to use a Signature permission on the Sensor Modules broadcast Intents and define a new receiver for events from third party applications. This would allow to deeply check these potential malicious events with only a small impact at the over-all performance.

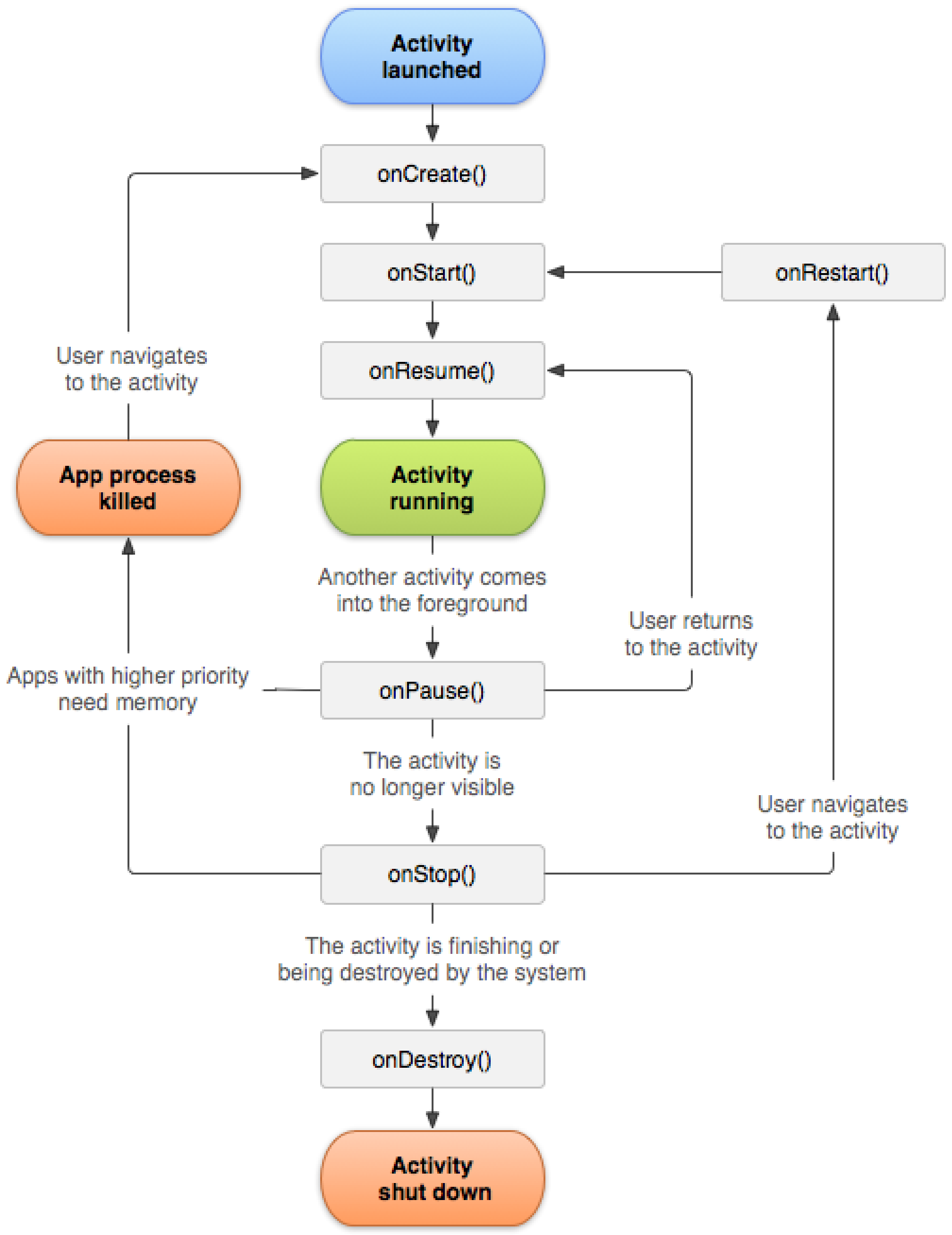
After the implementation of the Transformation Manager, the recovery process and database of the *subscriberChannelHashMap* has to be redefined. Otherwise there might be problems with too many transformation requests at the restore process.

Performance: In order to improve the sending performance it might be useful to only allow subscribing to leaves of the event tree. This would reduce the complexity to derive channels out of the event tree and the effort sending events to subscribers. Subscribing to nodes can be obtained, by raising the complexity of the subscription process. A subscription to the ROOT_CHANNEL would automatically subscribe to all leaves and all later advertised event types. The unsubscription process would be more complex as well. Also one has to keep in mind that the *advertisementHashMap* has to be stored in a persistent way.

The slow implementation to check if a publisher should receive a Start or Stop event should be improved.

Appendix

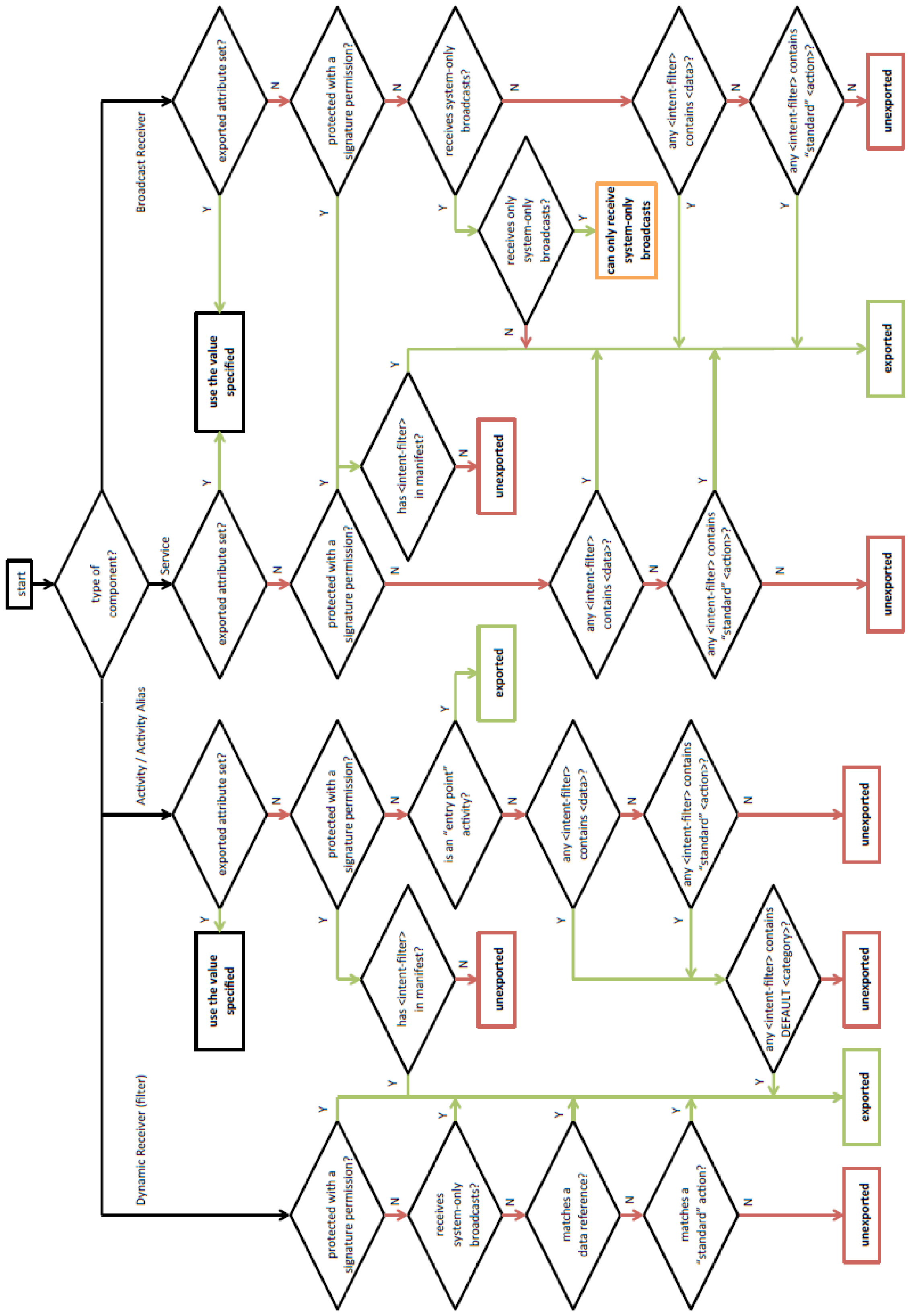
Appendix 1: Important state paths of an Activity[97]



Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

[97] Android (2013), http://developer.android.com/reference/android/app/Activity.html, 21-April-2013

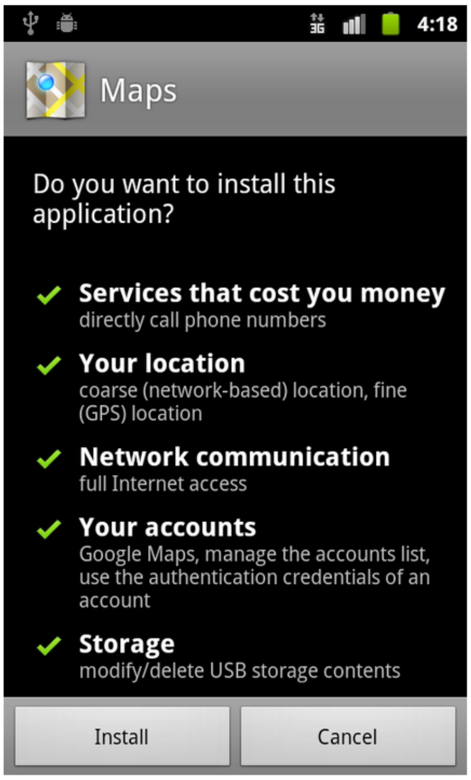
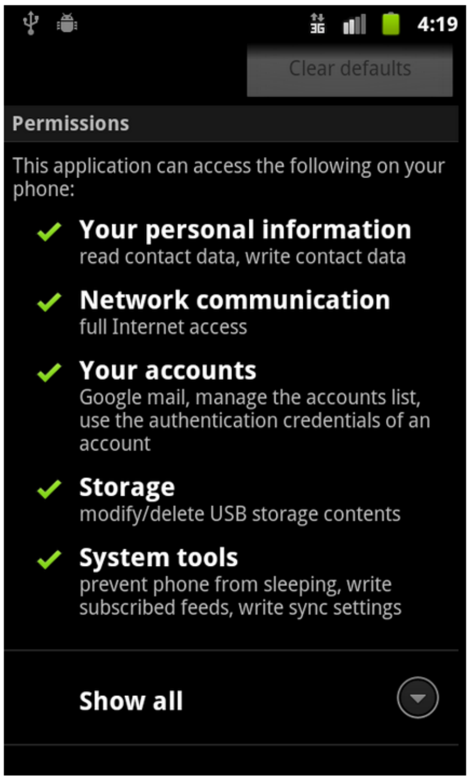
Appendix 2: Heuristic for when Components should be exported[98]



start
type of component?
Broadcast Receiver
Service
Activity / Activity Alias
Dynamic Receiver (filter)
exported attribute set?
use the value specified
protected with a signature permission?
receives system-only broadcasts?
receives only system-only broadcasts?
can only receive system-only broadcasts
has <intent-filter> in manifest?
is an "entry point" activity?
any <intent-filter> contains <data>?
any <intent-filter> contains "standard" <action>?
any <intent-filter> contains DEFAULT <category>?
receives system-only broadcasts?
matches a data reference?
matches a "standard" action?
exported
unexported
Y
N

98 Kantola et al. (2012)

Appendix 3: Permission dialog at application installation[99]

Permissions at Application Install -- Google Maps	Permissions of an Installed Application -- gMail
	

4:18

Maps

Do you want to install this application?

✓ **Services that cost you money**
directly call phone numbers

✓ **Your location**
coarse (network-based) location, fine (GPS) location

✓ **Network communication**
full Internet access

✓ **Your accounts**
Google Maps, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

4:19

Permissions

This application can access the following on your phone:

✓ **Your personal information**
read contact data, write contact data

✓ **Network communication**
full Internet access

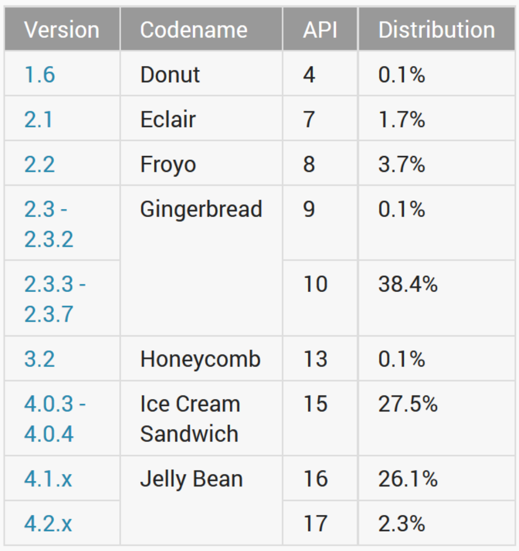
✓ **Your accounts**
Google mail, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

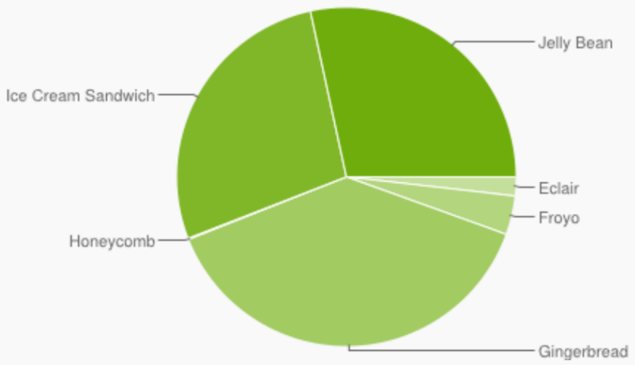
✓ **System tools**
prevent phone from sleeping, write subscribed feeds, write sync settings

[99] Android (2013a), http://source.android.com/tech/security/index.html, 16-May-2013

Appendix 4: Android versions market share[100]



Version	Codename	API	Distribution
1.6	Donut	4	0.1%
2.1	Eclair	7	1.7%
2.2	Froyo	8	3.7%
2.3 - 2.3.2	Gingerbread	9	0.1%
2.3.3 - 2.3.7		10	38.4%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	27.5%
4.1.x	Jelly Bean	16	26.1%
4.2.x		17	2.3%



Data collected during a 14-day period ending on May 1, 2013.
Any versions with less than 0.1% distribution are not shown.

100 Android (2013), http://developer.android.com/about/dashboards/index.html, 02-May-2013

Appendix 5: Source code for a sample application that uses myHealthAssistant

```
package com.example.myhealthhubsetpackage.performance.sample;

import de.tudarmstadt.dvs.myhealthhub.channels.AbstractChannel;
import de.tudarmstadt.dvs.myhealthhub.events.Event;
import de.tudarmstadt.dvs.myhealthhub.events.management.Subscribe;
import de.tudarmstadt.dvs.myhealthhub.events.management.Unsubscribe;
import de.tudarmstadt.dvs.myhealthhub.events.sensorreadings.SensorReadingEvent;
import de.tudarmstadt.dvs.myhealthhub.events.sensorreadings.cardiovascular.HeartRateEvent;
import de.tudarmstadt.dvs.myhealthhub.events.sensorreadings.physical.AccSensorEvent;
import android.os.Bundle;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.util.Log;
import android.view.Menu;
import android.view.View;

public class Sample extends Activity {

        private final String root = SensorReadingEvent.READING_EVENT;
        private final String acc = SensorReadingEvent.ACCELEROMETER;
        private final String hr = SensorReadingEvent.HEART_RATE;

        //used to publish events via myHealthAssistant
        private String myHealthAssistantReceiver = AbstractChannel.RECEIVER;
        private String myHeathAssistantPackageName = "de.tudarmstadt.dvs.myhealthhub";

        private IntentFilter readingChannelRoot = new IntentFilter(root);
        private IntentFilter readingChannelAccelerometer = new IntentFilter(acc);
        private IntentFilter readingChannelHeartRate = new IntentFilter(hr);

        private ReadingEventReceiver mReadingEventReceiver = new ReadingEventReceiver();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sample);
        //register for implicit broadcast
        registerReceiver(mReadingEventReceiver, readingChannelHeartRate);
        subscribe();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_sample, menu);
        return true;
    }

    public void subscribe(){
        Subscribe subscribe = new Subscribe(
                        "eventID",
                        "timestamp",
                        "producerID",
                        getPackageName(),
                        hr); //subscriber channel

        Intent intent = new Intent();
        intent.putExtra(Event.PARCELABLE_EXTRA_EVENT_TYPE, subscribe.getEventType());
        intent.putExtra(Event.PARCELABLE_EXTRA_EVENT, subscribe);
        //used for implicit delivery
        intent.setAction(AbstractChannel.MANAGEMENT);
        //used for explicit delivery to myHealthAssistant
        intent.setPackage(myHeathAssistantPackageName);
        sendBroadcast(intent);
    }
    public void unsubscribe(){
        Unsubscribe unsubscribe = new Unsubscribe(
                        "eventID",
                        "timestamp",
                        "producerID",
                        getPackageName(),
                        hr);

        Intent intent = new Intent();
        intent.putExtra(Event.PARCELABLE_EXTRA_EVENT_TYPE, unsubscribe.getEventType());
        intent.putExtra(Event.PARCELABLE_EXTRA_EVENT, unsubscribe);
        //used for implicit delivery
        intent.setAction(AbstractChannel.MANAGEMENT);
        //used for explicit delivery to myHealthAssistant
        intent.setPackage(myHeathAssistantPackageName);
        sendBroadcast(intent);
    }

    //Sent an event to myHealthAssistant in order to publish it
    public void injectEvent(View v){
        AccSensorEvent event = new AccSensorEvent(
                        "eventID",
                        "timestamp",
                        "producerID",
                        "sensorType",
                        "measurementTime",
                        0, 1, 2, 3, 4, 5);

        Intent i = new Intent();
        i.putExtra(Event.PARCELABLE_EXTRA_EVENT_TYPE, event.getEventType());
        i.putExtra(Event.PARCELABLE_EXTRA_EVENT, event);
        //implicit resolution to eventreceiver in myHealthAsssistant
        i.setAction(myHealthAssistantReceiver);
        //used for explicit delivery to myHealthAssistant
        i.setPackage(myHeathAssistantPackageName);
        sendBroadcast(i);
    }

    public class ReadingEventReceiver extends BroadcastReceiver{
        @Override
        public void onReceive(Context context, Intent intent) {
                Event evt = intent.getParcelableExtra(Event.PARCELABLE_EXTRA_EVENT);

                if(evt.getEventType().equals(HeartRateEvent.EVENT_TYPE)){
                        Log.d("sampleApp", "Received heart rate event");
                }
                if(evt.getEventType().equals(AccSensorEvent.EVENT_TYPE)){

                        Log.d("sampleApp", "Received acc sensor event");
                }
        }
    }

        @Override
        protected void onDestroy() {
                // TODO Auto-generated method stub
                super.onDestroy();
                unregisterReceiver(mReadingEventReceiver);
                unsubscribe();
        }
}
```

List of Figures

List of Tables

Bibliography

Android (2013)	**Android Developers (2013)** http://developer.android.com/index.html.
Android (2013a)	**Android Open Source Project (2013)** Tech Info http://source.android.com/tech/security/index.html.
Burns (2009)	**J. Burns (2009)** Mobile Application Security on Android, Black Hat, 2009.
Chin et al. (2011)	**E. Chin, A. P. Felt, K. Greenwood, and D. Wagner (2011)** Analyzing inter-application communication in Android. In Proceedings of the 9th international conference on Mobile systems, applications, and services (pp. 239-252). ACM.
Devmaze (2013)	**Devmaze (2013)** Android components' lifetime. http://devmaze.wordpress.com/2011/07/17/android-components-lifetime/
Eugster et al. (2003)	**P. T. Eugster, P. A. Felber, R. Guerraoui, and A.M. Kermarrec (2003)** The many faces of publish/subscribe. ACM Computing Surveys (CSUR), 35(2), 114-131.
Kantola et al. (2012)	**D. Kantola, E. Chin, W. He, and D. Wagner (2012)** Reducing Attack Surfaces for Intra-Application Communication in Android, SPSM 2012, October 19, 2012, Raleigh, North Carolina, USA.
Seeger et al. (2011)	**C. Seeger, A. Buchmann, and K. Van Laerhoven (2011)** Wireless Sensor Networks in the Wild: Three Practical Issues after a Middleware Deployment, MidSens 2011, December 12th, 2011, Lisbon, Portugal.
Seeger et al. (2012)	**C. Seeger, A. Buchmann, and K. Van Laerhoven (2012)** An Event-based BSN Middleware that supports Seamless Switching between Sensor Configurations, IHI 2012, January 28-30, 2012, Miami, Florida, USA.
Seeger et al. (2013)	**C. Seeger, K. Van Laerhoven, J. Sauer, and A. Buchmann (2013)** A Publish/Subscribe Middleware for Body and Ambient Sensor Networks that Mediates between Sensors and Applications, accepted to ICHI 2013, September 9-11, 2013, Philadelphia, Pennsylvania, USA.
Shannon et al. (1948)	**C. E. Shannon, and W. Weaver (1948)** A mathematical theory of communication.
Vennon (2010)	**M. T. Vennon (2010):** Android Malware: Spyware in the Android Market. Technical report, SMobile Systems, March 2010.