NTNU – Trondheim
Norwegian University of
Science and Technology

An API to Wi-Fi Direct Using Reactive Building Blocks

Erlend Bjerke Gabrielsen

Master of Telematics - Communication Networks and Networked
Submission date: June 2012
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Frank Kraemer, ITEM
Harald Viste, Pixavi

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Name of student: Erlend Bjerke Gabrielsen

Wi-Fi Direct is a networking technology supported by an increasing number of devices. It can be used to establish ad-hoc, peer-to-peer communication between devices without other networking infrastructure. In this master thesis, Arctis building blocks should be developed that support Wi-Fi Direct. The blocks should be designed so that applications using Wi-Fi Direct can be created easier, and without detailed knowledge of the particularities of these technologies. Different solutions should be discussed, and the effectiveness of the building blocks and the effect on application development should be shown by one or more examples.

Assignment given: 23.01.2012

Supervisor: Peter Herrmann, Professor, ITEM
Co-supervisor: Frank Alexander Kraemer, Ph.D., ITEM

Abstract

Implementing unfamiliar functionalities in smartphone applications can be a difficult and a tedious task. Owing to the fact that the Application Programming Interfaces (APIs) do not have a formal way of representing the sequence of events may be one reason. This thesis describes the development process of various Arctis building blocks based on Android's API of Wi-Fi Direct. The objective of these blocks was to simplify the implementation of Wi-Fi Direct by confining a predictable sequence of events.

An Android application was developed in order to test the functionalities, and to validate the prospects of portability for the various building blocks. The work resulted in a construction of three main building blocks, where each of them is responsible for a Wi-Fi Direct related function. Developers will be able to seamlessly utilize the Wi-Fi Direct functionality by combining and implementing these building blocks into their own applications.

Sammendrag

Implementasjon av ukjente funksjonaliteter i smarttelefonapplikasjoner kan være vanskelig og tidkrevende. En av grunnene til dette kan være at programmeringsgrensesnitt ikke har noen formell måte å representere rekkefølgen av begivenheter på. Denne hovedoppgaven beskriver utviklingsprosessen av forskjellige Arctis-byggeklosser basert på Androids programmeringsgrensesnitt av Wi-Fi Direct. Målet med hovedoppgaven var å enkeliggjøre implementasjonen av Wi-Fi Direct ved å begrense rekkefølgen av begivenheter, slik at det blir mer forutsigbart.

En Android applikasjon ble utviklet slik at funksjonalitetene kunne testes ut, i tillegg til å kunne validere mulighetene for å flytte byggeklossene fra et sted til et annet. Arbeidet resulterte i konstruering av tre hovedbyggeklosser, hvor hver kloss er ansvarlig for sin Wi-Fi Direct relaterte funksjon. Ved å kombinere og implementere disse byggeklossene inn i deres egene applikasjoner, vil utviklere ha muligheten til å enkelt utnytte funksjonaliteten til Wi-Fi Direct.

Preface

This thesis documents the work I have done as part of the master theses at Norwegian University of Science and Technology (NTNU). The work was done at Faculty of Information Technology, Mathematics and Electrical Engineering under the Department of Telematics.

I would like to thank my supervisor Peter Herrmann, and my co-supervisor Frank Alexander Kraemer for the guidance during the semester, and for showing initiative and interest in my work. In addition to my supervisors, I had support from a commercial actor. I would like to thank the employees at Pixavi for the guidance with establishing specific use-cases in order to have a foundation for modeling the Arctis blocks.

Erlend Bjerke Gabrielsen
June 2012

Contents

1 Introduction **1**

1.1 Simplicity vs. functionality 2

1.2 Investigating the API 2

2 Background **5**

2.1 Wi-Fi 5

2.2 Wi-Fi Direct 6

2.2.1 Security 6

2.2.2 Group owner 8

2.2.3 Key mechanisms 8

2.2.4 Optional capabilities 9

2.2.5 Power management 11

2.3 Wi-Fi Direct in Android 13

2.3.1 API specific components 13

2.3.2 The Wi-Fi Direct API 15

3 Methodology **25**

3.1 The choice of method 25

3.2 The development process 26

3.3 Connection issue 28

3.4 Development environment 29

4 System Implementation Design **31**

4.1 Implementation alternatives 31

4.2 Design choices 33

5 API Building Blocks **41**

5.1 The Wifi Direct Receive block 42

5.1.1 Description of the Wifi Direct Receive block 43
5.1.2 Analysis of the Wifi Direct Receive block 44
5.2 The Wifi Direct Discover Peers block 44
5.2.1 Description of the Wifi Direct Discover Peers block 45
5.2.2 Analysis of the Wifi Discover Peers block 47
5.3 The Wifi Direct Connect block 48
5.3.1 Description of the Wifi Direct Connect block 48
5.3.2 Analysis of the Wifi Direct Connect block 52
5.4 Combining the API blocks 52
5.4.1 Description of the Wifi Direct Service block 57
5.4.2 Analysis of the Wifi Direct Service block 58
5.4.3 The Wifi Direct Responder and the Wifi Direct Initiator . . 59
5.5 Evolvement of the Wifi Direct Service block 60
5.5.1 The Wifi Direct 1 block 61
5.5.2 The Wifi Direct 3 block 63
5.5.3 The Wifi Direct 4 block 64

6 Example Application 67

6.1 Application building blocks 68
6.1.1 The Wifi Direct Application Overview block 68

7 Discussion 79

7.1 Findings 79
7.2 Limitations 81
7.3 Further Work 81

8 Conclusion 83

References 85

List of Figures

1.1 The ESM of the Discover Peers block 3
1.2 A snippet of a block that encapsulates the Discover Peers block . . 4

2.1 Examples of the PIN method's sequence 7
2.2 P2P group configurations 9
2.3 Example of a smartphone with a concurrent connection 10
2.4 The operation of the OPS protocol 11
2.5 Example of a NoA operation 12
2.6 A graph based on the distribution of Google Play's users 13
2.7 A UML class diagram of the API of Wi-Fi Direct 16
2.8 A state machine of the initialization process 18
2.9 A state machine of the discovery process 19
2.10 A state machine of the connection process 21
2.11 A state machine of the disconnection process 22

3.1 The waterfall model 26
3.2 The iterative model 26
3.3 An overview of the development process 27

4.1 Illustration of the system 32
4.2 An informal sequence diagram which illustrates the media stream . 33
4.3 Example with a list of available peers 36
4.4 An informal sequence diagram showing an example of the 38
4.5 An informal sequence diagram showing the discovery and 39

5.1 The internal structure of the Wifi Direct Receive block 42
5.2 The ESM of the Wifi Direct Receive block 44
5.3 The left side of the Wifi Direct Discover Peers block 45

5.4 The right side of the Wifi Direct Discover Peers block 46
5.5 The internal structure of the Discover Peers block 47
5.6 The ESM of the Wifi Discover Peers block 47
5.7 The left side of the Wifi Direct Connect block 49
5.8 The right side of the Wifi Direct Connect block 50
5.9 The internal structure of the Connect, Cancel Connect and 51
5.10 The ESM of the Wifi Direct Connect block 53
5.11 The left side of the Wifi Direct Service block 54
5.12 The middle part of the Wifi Direct Service block 55
5.13 The right side of the Wifi Direct Service block 56
5.14 The ESM of the Wifi Direct Service block 58
5.15 The Wifi Direct Responder block 60
5.16 The Wifi Direct Initiator block 61
5.17 A snippet of some of the Wifi Direct 1 block's input pins 62
5.18 A snippet of the Wifi Direct 1 block 63
5.19 A snippet of the Wifi Direct 1 block 64
5.20 A snippet of the Wifi Direct 3 block's input pins 65
5.21 A snippet of the Wifi Direct 4 block 66

6.1 The Wifi Direct System block 68
6.2 The left side of the Wifi Direct Application Overview block 69
6.3 The middle part of the Wifi Direct Application Overview block . . 70
6.4 The right side of the Wifi Direct Application Overview block 71
6.5 The application is initialized 72
6.6 The discovery initiation process with two possible outcomes 73
6.7 A connection request is received 74
6.8 The devices are connected 75
6.9 Group information is displayed on the UI 76
6.10 The Open Camera button has been pushed on the group owner . . 77

7.1 Example of an ANR dialog 80

List of Tables

2.1 The request methods specified by the API 17

Acronyms

ANR Application Not Responding

AP Access Point

API Application Programming Interface

ESM External State Machine

GUI Graphical User Interface

HD High-Definition

HDMI High-Definition Multimedia Interface

IEEE Institute of Electrical and Electronics Engineers

LAN Local Area Network

LED Light-Emitting Diode

MAC Media Access Control

MDE Model-Driven Engineering

NFC Near Field Communication

NoA Notice of Absence

OMG Object Management Group

OPS Opportunistic Power Save

P2P Peer-to-Peer

PBC Push Button Configuration

PIN Personal Identification Number

QoS Quality of Service

SSID Service Set Identifier

SDK Software Development Kit

SDL Specification and Description Language

TCP Transmission Control Protocol

UI User Interface

UML Unified Modeling Language

URI Uniform Resource Identifier

USB Universal Serial Bus

WECA Wireless Ethernet Compatibility Alliance

WFA Wi-Fi Alliance

WLAN Wireless Local Area Network

WPAN Wireless Personal Area Network

WPS Wi-Fi Protected Setup

Chapter 1

Introduction

Android provides several APIs to facilitate interaction between software components and the operating system. Each API contains a set of functions that together form its functionality.

Some of these APIs contains time-consuming functions that can take an arbitrarily time to run, which must be executed simultaneously with other tasks. Using them is known as asynchronous programming and is essential in Android development where reactiveness is a requirement. Without asynchronous programming, only one task can be executed at once, resulting in the application to be blocked until the current task is finished. However, asynchronous APIs are complicated. The sequence of executions and the awareness of the application's state during the reception of messages are important factors developers need to take into consideration when using such APIs. Modeling the application's behavior should therefore be a crucial part of the development.

By dividing an API into different generic templates based on predefined use-cases, we wanted to find out if an implementation of the API's functions could be made easier to achieve. Hence, with the assistance of Arctis Software Development Kit (SDK) we constructed building blocks to model such templates based on the *android.net.wifi.p2p* API. For simplicity, these building blocks will be referred to as *API blocks* and the API will be referred to as the *Wi-Fi Direct API* henceforward in this thesis.

This API provides the possibility to crate Peer-to-Peer (P2P) connections using Wi-Fi Direct (see Sect. 2.2 for more information of Wi-Fi Direct). It includes the following properties of interest:

- The API is asynchronous, which makes it quite complex and more dependable on model specifications.
- The Wi-Fi Direct technology is novel and most developers are therefore unacquainted with this particular API.
- The API includes a variety of potential use-cases.

1.1 Simplicity vs. functionality

Even though individual functions within an API are well documented, they can still be misinterpreted. Developers could for instance entirely understand a function's property, but not how it is supposed to perform together with other functions to form a specific service. Hence, by implementing functionalities only based on an API specification requires multiple trials and errors before a successful implementation is accomplished. This is because the Java compiler doesn't specify the sequence of API operations, and it is left to developers to decide this. However, if they rather implement building blocks based on use-cases from the API, the complexity of adapting it is drastically reduced resulting in much time saved.

There is a tradeoff between an API building block's functionality and the simplicity of implementing it. If a building block encapsulates too much of an API's functioning into one sequence of events, it will be fairly simple to use it. However, the variety of events is constricted to meet the building block's logical design. Hence, the block is only useful if developers seek the exact same sequence of events. On the other hand, if the block does not contain sufficient sequential logic there is no gained simplicity. This is because the lack of restrictions increases the probability of design flaws.

1.2 Investigating the API

The difficulty is to obtain a suitable amount of functionality for a particular API building block. By studying the API, we realized that some events are expected to happen in one particular sequence. Even though this sequence is specified by the documentation, constraints to follow it are not established by the API.

By for instance examining the *WifiP2pManager* class in the Wi-Fi Direct API [1], there is a request method named *discoverPeers* (see Sect. 2.3.2 for more details on request methods). The documentation of this method reveals the following [1]. «The function call immediately returns after sending a discovery request to the

framework. The application is notified of a success or failure to initiate discovery through listener callbacks onSuccess() or onFailure(int).» This means that the after the discoverPeers method has been executed, the application should expect either an *onSuccess* or an *onFailure* callback to be triggered. This behavior is reflected in Fig. 1.1, where a building block named *Discover Peers* puts a constrain to the sequence. See Fig. 5.5 in Sect. 5.2.1 for the internal structure of this block. The block is initiated by the *start* pin being triggered. When the block is in state *active*, it will execute the discoverPeers method and wait for an onSuccess or an onFailure to be triggered. When this happens, the corresponding output pin is triggered.

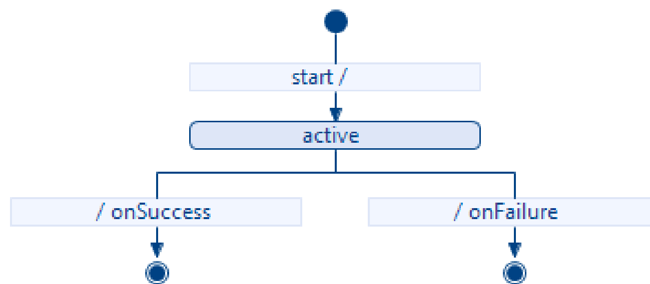


Figure 1.1: The ESM of the Discover Peers block

This simple External State Machine (ESM) will together with Fig. 5.5 enhance the understanding of the expected sequence of events. The reason for this is that a written documentation can more easily suffer from being misinterpreted than graphical illustrations. The documentation of the discoverPeers method carries on by stating the following [1]. «Register for WIFI_P2P_PEERS_CHANGED_ACTION intent to determine when the framework notifies of a change as peers are discovered.» The application must therefore be ready to receive an intent from the framework if the onSuccess callback was triggered. See Sect. 2.3.1 for a brief description on intents.

By encapsulating the Discover Peers block in another block, additional behavior can be added. This is shown in Fig. 1.2. First, a broadcast receiver is registered in the *registerBroadcastReceiver* operation. This broadcast receiver have the *WIFI_P2P_PEERS_CHANGED_ACTION* intent action in its intent filter. In the *initDiscoverPeers* operation, an instance of the *DiscoverPeersInfo* class is returned. This class includes two public variables of the data types *Channel* and *WifiP2pManager*. See Sect. 2.3.1 for a short explanation on broadcast receivers.

In order to be in consistence with the documentation, if the onSuccess pin is triggered at the Discover Peers block, an event identified as *WIFI_P2P_PEERS_CHA-*

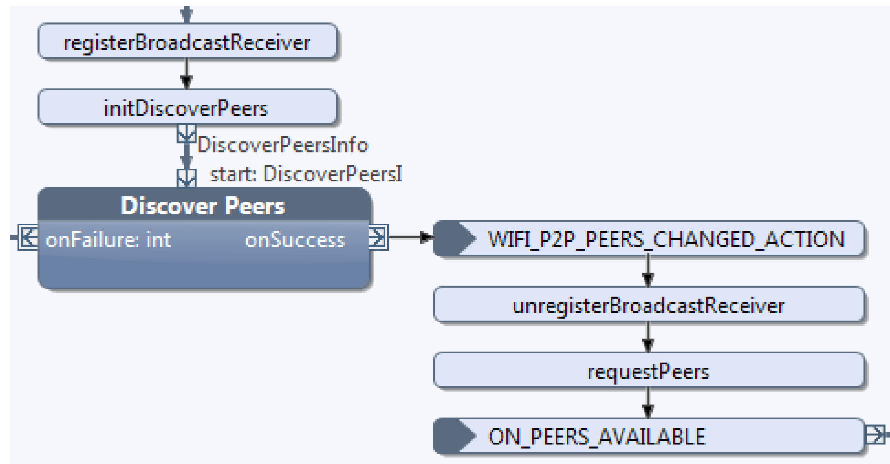


Figure 1.2: A snippet of a block that encapsulates the Discover Peers block to add behavior

NGED_ACTION will cause the outer block to wait for the broadcast intent with the same name to be received from the Android framework.

Figure 1.2 also shows that some additional behavior is added to the block subsequent to the WIFI_P2P_PEERS_CHANGED_ACTION event. This is in agreement with the following statement from the documentation of the discoverPeers method [1]. «Upon receiving a WIFI_P2P_PEERS_CHANGED_ACTION intent, an application can request for the list of peers using requestPeers(WifiP2pManager-.Channel, WifiP2pManager.PeerListListener).» Therefore, after the event has been triggered, the previously registered broadcast receiver is unregistered in the *unregisterBroadcastReceiver* operation and the current list of peer devices is requested in the *requestPeers* operation. This method has a listener callback that is triggered when the requested list is available. The application should therefore expect this to happen. This is shown in Fig. 1.2 by the ON_PEERS_AVAILABLE event.

Chapter 2

Background

This chapter provides an introduction to the technologies and concepts that is relevant for this thesis. Wi-Fi Direct is a new technology and is therefore thoroughly elaborated, both individually and together with Android. It is assumed that the reader has a basic knowledge of the Android platform and the Arctis framework. However, some Android specific components that are directly related to Android's Wi-Fi Direct API will be elaborated.

2.1 Wi-Fi

Wi-Fi is a marketing brand for products that are based on the IEEE 802.11 working group and is often mistakenly referred to as *wireless fidelity* [2]. Wi-Fi provides wireless access to LANs by using radio frequency as the transport mean. The marketing success of the Wi-Fi brand is tremendous and roughly ten per cent of all people in the world use Wi-Fi for Internet connectivity [3].

When the 802.11 standard was introduced by IEEE, it was interpreted by the various manufactures in different ways [2]. This led to a variety of 802.11 compatible devices which did not interoperate with each other. To prevent this expansion of incompatible wireless devices among the different vendors, a group called Wireless Ethernet Compatibility Alliance (WECA) was formed in August 1999. This group is currently known as the Wi-Fi Alliance (WFA) after they changed their name in October 2002. The WFA's role is to perform various tests for certification of wireless devices to be Wi-Fi compliant. As a result, the Wi-Fi certified devices are able to interoperate with each other.

2.2 Wi-Fi Direct

In October 2010, the WFA introduced a new network technology called Wi-Fi Direct [4]. It allows for wireless devices to have direct P2P connectivity without the need of any traditional network infrastructure or Access Point (AP) between them [3]. It is built upon the well-established Wi-Fi specification, which means that it belongs to the IEEE 802.11 working group. Wi-Fi Direct support data rates from all existing IEEE 802.11 protocols except from IEEE 802.11b. This results in an efficient utilization of the wireless bandwidth.

Because of its similarities with Bluetooth, a common misinterpretation is to relate this network type to Wireless Personal Area Network (WPAN). But WPAN lies within the IEEE 802.15 working group's area and does not comprise Wi-Fi Direct.

Wi-Fi Direct certified devices support connectivity for legacy Wi-Fi equipment, which means that an ordinary Wi-Fi device can discover and connect to Wi-Fi Direct groups. A Wi-Fi Direct group is a set of devices that are connected together via Wi-Fi Direct to form an ad-hoc network (details of group formation is given in Sect. 2.2.3). The group owner will appear as an ordinary AP for legacy equipment (see Sect. 2.2.2 for information about the group owner).

2.2.1 Security

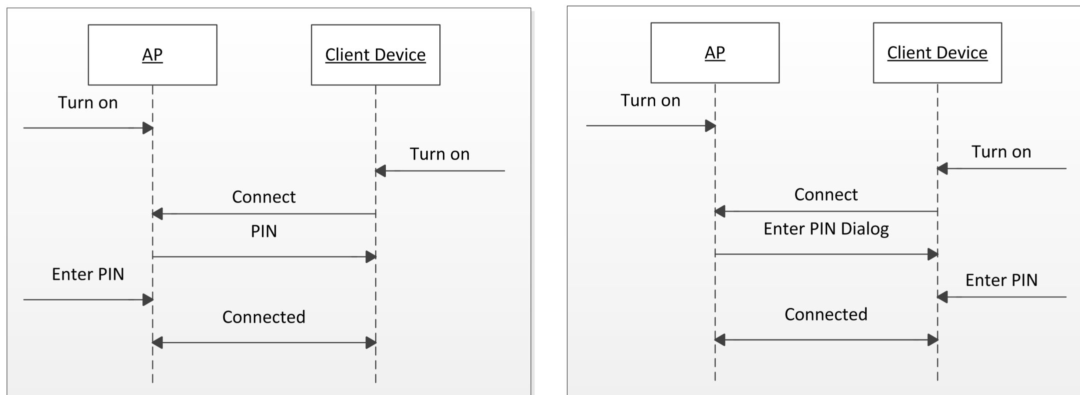
Wi-Fi Direct certified devices are using Wi-Fi Protected Setup (WPS) when they are connecting with each other. WPS is a setup mechanism that let users add Wi-Fi compatible devices to Wi-Fi networks more seamlessly, and with the same level of security as with traditional manual Wi-Fi setup procedures. The main reason for this is to be able to configure Wi-Fi networks without the knowledge of the underlying technologies or processes involved in the setup procedure [5]. There are four types of setup methods users can choose to implement [6]:

- The Personal Identification Number (PIN) method.
- The Push Button Configuration (PBC) method.
- The Near Field Communication (NFC) method.
- The Universal Serial Bus (USB) method.

The AP, or the group owner in Wi-Fi Direct is required to offer both the PIN and the PBC methods, while the client devices must at least offer the PIN method. The last two methods are optional.

The PIN method

The reason for having a PIN is to ensure that the right device is added to the network, and to prevent accidental or malicious attempts of adding unintended devices [5]. A PIN can be dynamically generated, and distributed to the connecting client device from the AP. In this way, the PIN will be shown at the connecting device's display but is required to be entered at the AP's User Interface (UI) (see Fig. 2.1a). Another way is to have a fixed PIN placed on the AP. The fixed PIN is then required to be entered on the connecting device's UI (see Fig. 2.1b).



(a) PINs are dynamically generated

(b) The fixed PIN's sequence

Figure 2.1: Examples of the PIN method's sequence

The PBC method

This method is realized by having an interaction with both the AP and the client device when users want to add a device to the network. This interaction will typically be a button being pushed (physical or virtual) at both the AP and the client device [5]. In the time between these buttons are pushed, there is a possibility for unintended devices within the range to join the network. This could possibly result in a vulnerability to the network and should be taken into consideration when choosing method for the setup procedure.

The NFC and the USB method

In order to employ the NFC method, both the client device and the AP must be equipped with the NFC technology [6]. By bringing the client device close enough to be within NFC range of the AP, the authentication can be performed by using the NFC channel. With the USB method, the client uses a USB flash drive to transfer the authentication data manually between the client device and the AP.

2.2.2 Group owner

In every P2P group there is a device that will be in charge. This device is called the group owner and will act as an AP on behalf of the other group members. This includes governing the group's initiation and termination process, in addition to controlling which devices that are allowed to participate in the group.

All Wi-Fi Direct certified devices must be able to act as a group owner [3]. In addition, they must be able to negotiate which of the connecting devices that will become the group owner. Hence, if an automatic group formation takes place, a negotiation between the connecting devices are performed to determine the device that will become the group owner. Otherwise, the device that autonomously initiated the P2P group will act as the group owner. A Wi-Fi Direct device can either have a temporary or persistent connection over a longer period of time. It is the group owner that decides what kind of connection the group is going to have.

2.2.3 Key mechanisms

There are several key mechanisms specified in the WFA's P2P specification. Some of them are mandatory while others are optional [3]:

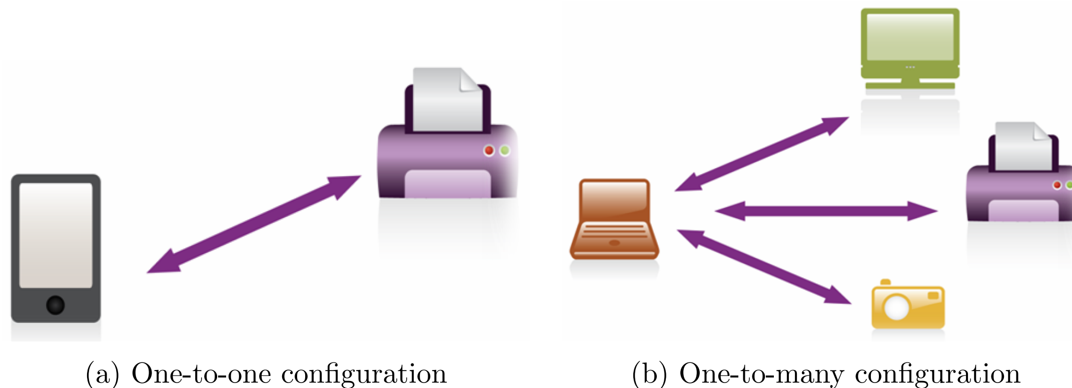
- Mandatory mechanisms
  - Device discovery
  - Group formation
  - Client discovery
- Optional mechanisms
  - Service discovery
  - Invitation

Device discovery and group formation

Device discovery is used to detect other Wi-Fi Direct compatible devices within the reach. When a device is discovered, a connection to the discovered device can be established. If the target device is already part of a P2P group, a request can be sent in order to join the particular group. Otherwise, a new group will be formed automatically when the connection request is initiated.

A group will always be formed irrespective of how many devices that are going

to be connected to each other. A group can even be formed by a single device alone. This is convenient if the device is going to provide a specific service (e.g. internet connectivity) to other devices. In addition, this is required if all the other devices are legacy equipment. A group can either be formed manually (with a single device alone) or automatically (when connecting multiple devices). A P2P group can both have a one-to-one and one-to-many configuration (see Fig. 2.2).



(a) One-to-one configuration (b) One-to-many configuration

Figure 2.2: P2P group configurations. Taken from the WFA [3]

Client discovery and service discovery

In order to find out what kind of devices that are connected to a P2P group, a *client discovery* is initiated. A smartphone can for instance inquire other devices within a group to find out if there exists a TV for displaying an image, or a printer for a printing request.

Instead of inquiring a specific device in order to perform something, a request for a specific service could be initiated. This is called *service discovery* and can be performed at any time, even prior to establishing a Wi-Fi Direct connection [3]. It is supported by higher layer applications such as UPnP and Bonjour.

2.2.4 Optional capabilities

The WFA's P2P specification introduces the following optional capabilities within Wi-Fi Direct [3]:

- Persistent groups
- Concurrent connection
  - Multiple groups
  - Cross-connection

- Managed device

Persistent groups

By storing the group information and credentials, persistent groups eliminates the process of WPS when a group is re-invoked [3]. This means that users don't need the additional interaction that comes with WPS, which is useful for devices that are frequently re-connected. It is the group owner that decides whether the group should be persistent or not. An example of a situation where persistent groups are valuable is when a laptop connects with a printer. With a persistent group, the only requirement is that the laptop is in range of the printer in order to use it.

Concurrent connections

A concurrent Wi-Fi Direct device can be connected to multiple P2P groups or external networks at the same time. Figure 2.3 shows an example of a concurrent connection. This figure illustrates a smartphone that is connected to two different Wi-Fi Direct groups and one external Wi-Fi network at the same time.

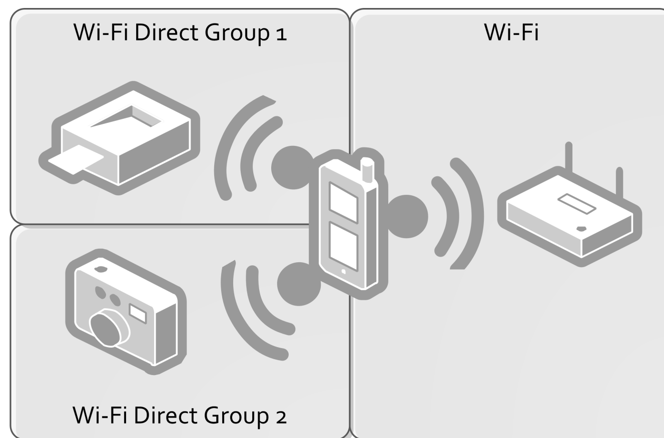


Figure 2.3: Example of a smartphone with a concurrent connection

To attain concurrent connections, a distinct Media Access Control (MAC) entity for each type of connection is needed [3]. This is realized by having multiple MAC entities on a single device. These entities are either physically or virtually separated. Thus, it is the number of MAC entities on a device that limits the number of concurrent connections it can incorporate.

When a device has a connection with an additional external network, it is characterized to have a cross-connection. The device will then be able to provide Internet connectivity for the other devices within the Wi-Fi Direct group(s) it is a part of. To achieve this, the device is required to act as the group owner in addition to

have sufficient MAC entities.

Managed device

An AP may be configured with capabilities to support management of Wi-Fi Direct devices to protect an enterprise infrastructure network [3]. The AP may monitor the connected Wi-Fi Direct devices and possibly expel, if out-of-policy behavior is detected. A Wi-Fi Direct device may implement managed device mechanisms to assist the AP in managing the Wi-Fi environment. This device will be able to receive service information from the AP and send useful information back. An AP can for instance have the option to only authenticate Wi-Fi Direct devices that has implemented the managed device mechanism.

2.2.5 Power management

A typical Wi-Fi Direct device is portable with a limited battery-lifetime. Hence, efficient use of power is important in Wi-Fi Direct to avoid draining of the device's battery. The WFA's P2P specification includes power management to minimize the power consumption regardless of the role or the state of the device within a P2P group [3]. Irrespective of these power management features, the power consumption depends on the settings and the interactions between the devices. Along with some adapted legacy power management mechanisms from Wi-Fi, two new power saving protocols are included in Wi-Fi Direct. These are the Opportunistic Power Save (OPS) and the Notice of Absence (NoA) [7]. The reason for adding these protocols is that the group owner behaves differently than the clients, which results in a power consumption that is significantly higher.

The OPS protocol

As the name implies, OPS is a protocol that opportunistically saves the group owner's power by going in sleep mode when all its associated P2P clients are sleeping [5]. Figure 2.4 shows the OPS protocol's operation.

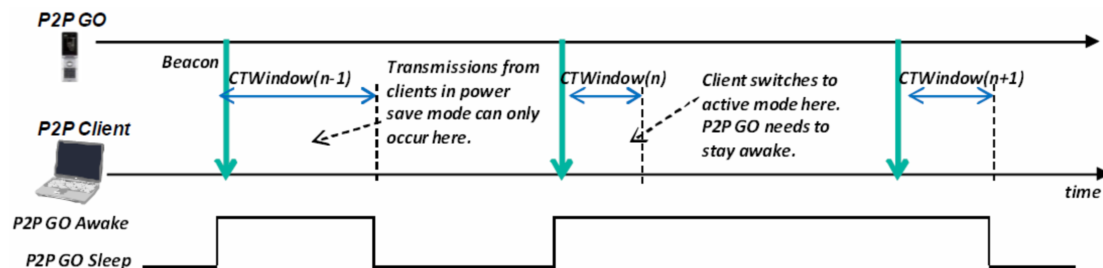


Figure 2.4: The operation of the OPS protocol. Taken from Campus-Mur et. al. [7]

The group owner specifies a limited presence period where the P2P clients are allowed to transmit. This time period is called the *CTWindow* and takes place when a beacon frame is sent from the group owner. If the group owner senses that a client is active at the end this time period, it will continue to stay awake. On the other hand, if no clients are active, the group owner will enter sleep mode until the next beacon frame is sent.

The possibility of entering sleep mode and save power for the group owner is being reduced as the number of clients in the group increases. This place a limitation on the total amount of power the OPS protocol is capable of saving in large P2P groups.

The NoA protocol

In contrast to the OPS protocol, the NoA protocol makes it possible for the group owner to save power regardless of the clients' state. Figure 2.5 shows an example of a NoA operation.

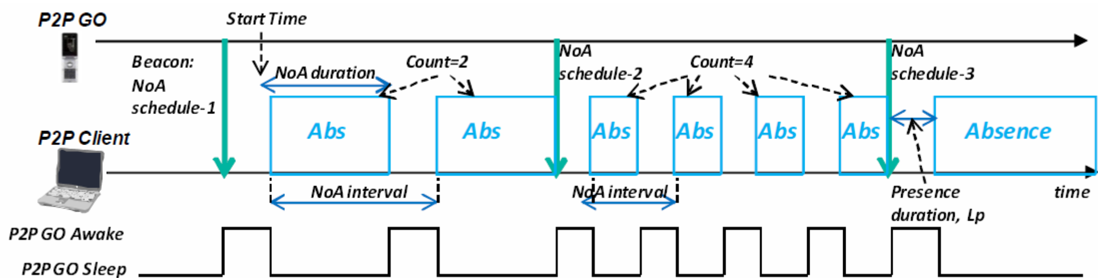


Figure 2.5: Example of a NoA operation. Taken from Campus-Mur et. al. [7]

Four different parameters contribute to schedule the group owner's absence:

- *Start time* determines the staring time of the next absence period.
- *Duration* regulates the duration of a specific absence period.
- *Interval* controls the time between the start of two consecutive absence periods.
- *Count* denotes the number of absence periods within a NoA schedule.

A beacon frame or a probe response frame will contain a NoA schedule. These frames will either start a new NoA schedule or update the current one. By omitting the signaling element in these frames, a NoA schedule is cancelled. The clients will always act in accordance with the latest advertised NoA schedule. To ensure Quality of Service (QoS), a client can request the group owner to be present at

certain intervals by a mechanism termed *P2P presence request/response handshake*.

2.3 Wi-Fi Direct in Android

Wi-Fi Direct is a new technology that has recently made its entry into the smartphones. Android 4.0 (Ice Cream Sandwich) was the first Android platform with built-in Wi-Fi Direct capabilities [8]. The first release of this platform was launched in October 2011, one year after the WFA's introduction of Wi-Fi Direct. By May 1, 2012 no more than roughly five per cent of the users who accessed Google Play operated on a device with an Android 4.0 - 4.0.3 platform [9]. This means that only a minority of today's Android users are able to utilize this technology. However, the reason for this small percentage is the low maturity of the platform, and the number of Android users with Wi-Fi Direct compatible platforms will certainly grow in the future. See Fig. 2.6 for a chart of the distribution based on the number of Android devices accessed Google Play within a 2 weeks period ending on May 1, 2012.

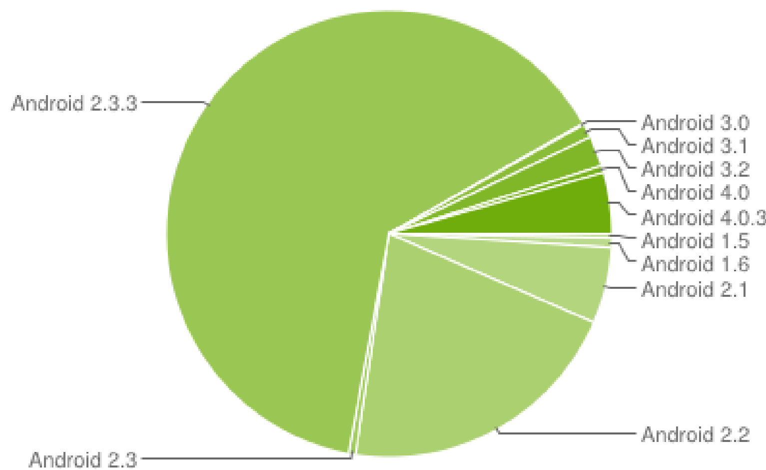


Figure 2.6: A graph based on the distribution of Google Play's users. Taken from the Android developer's guide [9]

2.3.1 API specific components

Intents are asynchronous messages that are sent to announce when some specific operations should be performed [10]. This could for instance be to initiate the web browser to open a web page. Intents can activate the following three main components of an Android application:

- *Activities*
- *Services*

- *Broadcast receivers*

In addition to announce operations to be done, intents can also be used to announce that certain events have happened. These messages (called broadcast intents) will be broadcasted to all relevant components. Applications can listen for broadcast intents and react on them by registering broadcast receivers.

A broadcast receiver is capable of responding to announcements, whether it originates from the system (e.g. when the battery is low or when the screen has been unlocked), from other applications, or from its own application [11]. Broadcast receivers are not directly connected with a UI, but they may trigger the UI to perform certain actions.

Intents can be divided in the following two groups [10]:

- *Explicit intents*
- *Implicit intents*

The difference between these groups is in how they address the target component. Explicit intents locate components by explicitly specifying the components name (e.g. *com.example.project*). These types of intents obviously require developers to know the exact name of the components, which is normally not the case for developers of other applications. Hence, intents within this group are typically not used for system-wide messaging.

Implicit intents on the other hand do not specify the target by its name. This means that the Android system needs another way of resolving the intent's receiving component. This is done by registering intent filters on the receiver component. The system will then test the intent filters against the broadcasted intents in order to find the best suitable receiving component.

There are three different tests an intent must go through, for successfully deliver an implicit intent to a receiving component [10]:

- *Action test*
- *Category test*
- *Data test*

Each of these tests involves comparing the information in the different fields of the intent with the intent filter at the receiving component. To pass the action

test, the action specified in the intent's action field must at least match one of the action elements listed at the receiver's intent filter [10]. To pass the category test, the receiver's intent filter must contain at least every category elements in the intent's category field.

Each data element in the data field can contain a Uniform Resource Identifier (URI) and a data type. There are different rules whether an element contains a URI, a data type, both a URI and a data type, or none of them. The Wi-Fi Direct API neither specifies a URI nor a data type for receiving broadcast intents. In this case, the test will be passed if the intent's data element contains neither a URI nor a data type.

2.3.2 The Wi-Fi Direct API

The Android API level 14 and higher incorporates the opportunity for applications to discover, connect and communicate by the use of Wi-Fi Direct [12]. Figure 2.7 shows a Unified Modeling Language (UML) class diagram of the Wi-Fi Direct API (the association with a crossed circle represents inner classes). This diagram shows that the *WifiP2pManager* is the primary class, which is composed of the following three main parts:

- *Listeners*
- *Request methods*
- *Intent actions*

Listeners

The message passing for Wi-Fi Direct in Android is asynchronous and the API specifies listener callback methods that are responsible for reacting to requests from the application. The following five different interfaces represent the various listeners:

- *ActionListener*
- *ChannelListener*
- *ConnectionInfoListener*
- *GroupInfoListener*
- *PeerListListener*

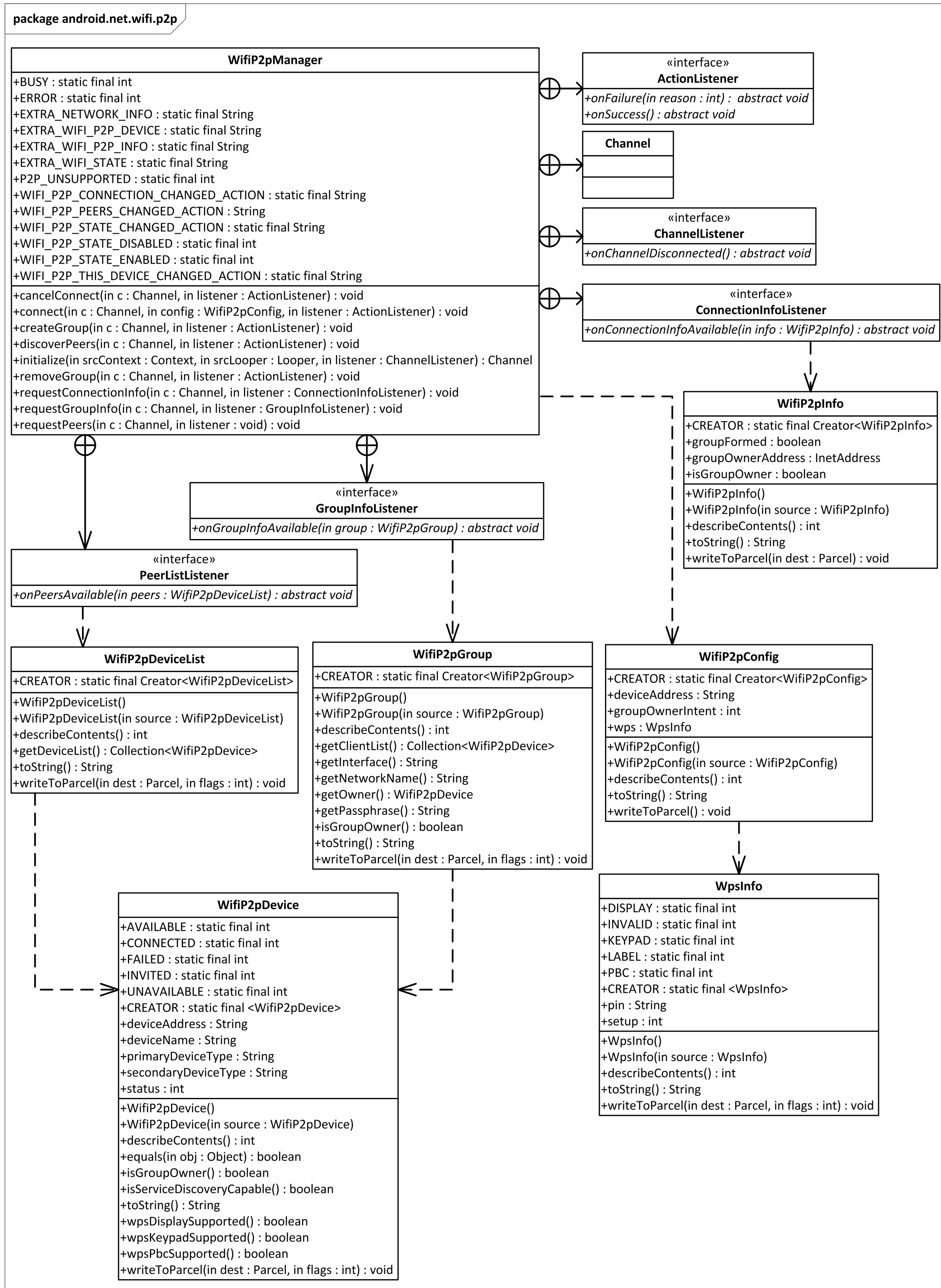


Figure 2.7: A UML class diagram of the API of Wi-Fi Direct

Each of these interfaces has callback methods which are triggered when a response is sent. The ActionListener's callback methods inform whether the operation was successful or not. In case of a failure, the callback will convey a constant to point

out the reason. This reason can be one of the following [1]:

- *ERROR* denotes that the reason was due to an internal failure.
- *P2P_UNSUPPORTED* indicates that Wi-Fi Direct is not supported by the current device.
- *BUSY* means that the framework is busy, and therefore unable to serve the request.

The ChannelListener's callback will be triggered if the channel gets disconnected from the framework [13]. The remaining listeners are triggered when some specific requested information is available.

Request methods

The API has defines nine different request methods (see Tab. 2.1). Some of them are required to be implemented, while others are optional. By using these methods, the application will be able to request the operating system to perform specific actions. Each of them will trigger asynchronous message requests and they should therefore be able to react when responses are sent. This is why each method includes a listener for callbacks.

Table 2.1: The request methods specified by the API. Taken from the Android API [1]

Public Methods	
void	cancelConnect(WifiP2pManager.Channel c, WifiP2pManager.ActionListener listener) Cancel any ongoing p2p group negotiation The function call immediately returns after sending a connection cancellation request to the framework.
void	connect(WifiP2pManager.Channel c, WifiP2pConfig config, WifiP2pManager.ActionListener listener) Start a p2p connection to a device with the specified configuration.
void	createGroup(WifiP2pManager.Channel c, WifiP2pManager.ActionListener listener) Create a p2p group with the current device as the group owner.
void	discoverPeers(WifiP2pManager.Channel c, WifiP2pManager.ActionListener listener) Initiate peer discovery.
WifiP2pManager.Channel	initialize(Context srcContext, Looper srcLooper, WifiP2pManager.ChannelListener listener) Registers the application with the Wi-Fi framework.
void	removeGroup(WifiP2pManager.Channel c, WifiP2pManager.ActionListener listener) Remove the current p2p group.
void	requestConnectionInfo(WifiP2pManager.Channel c, WifiP2pManager.ConnectionInfoListener listener) Request device connection info.
void	requestGroupInfo(WifiP2pManager.Channel c, WifiP2pManager.GroupInfoListener listener) Request p2p group info.
void	requestPeers(WifiP2pManager.Channel c, WifiP2pManager.PeerListListener listener) Request the current list of peers.

In the following, some informal state machines based on the Specification and Description Language (SDL) semantic are shown. They was created by the author and later used as a starting point for the API blocks. Thus, these state machines are not a part of the API, but a proposed sequence of events. We opted to have them in this section, since they facilitate a wider understanding of how the request methods relates to the other components of the API.

In order to implement a Wi-Fi Direct functionality in an application, a registration to the Wi-Fi framework is required [1]. This is realized by executing the *initialize* method in Tab. 2.1. All other request methods in the Wi-Fi Direct API depend on this registration. Hence, this must be the first Wi-Fi Direct operation to be performed. Figure 2.8 shows a proposed state machine of the initialization process. When the application enters the *initialized* state it should be able to discover other peer devices. The WIFI_P2P_STATE_CHANGED_ACTION, WIFI_P2P_THIS_DEVICE_CHANGED_ACTION and WIFI_P2P_CONNECTION_CHANGED_ACTION intent actions are explained later in this section.

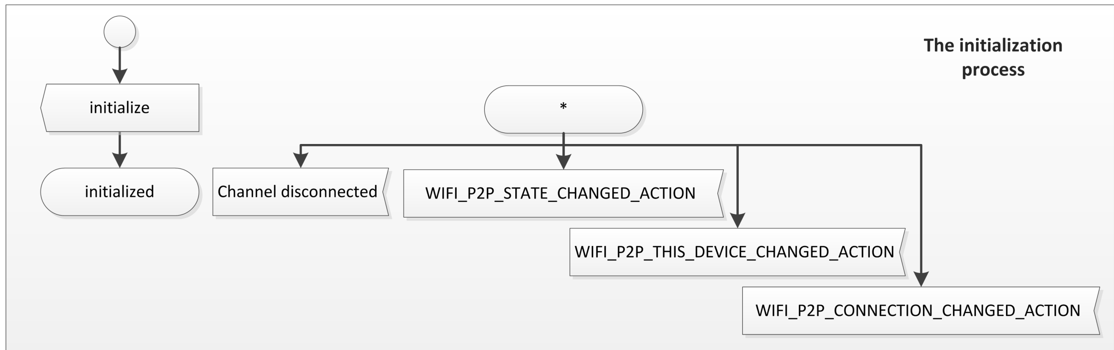


Figure 2.8: A state machine of the initialization process

To be able of finding peer devices, the application must execute the *discoverPeers* method. This operation initiates a peer discovery, which involves sending a request to the framework to scan for available peer devices. If the request is successfully achieved, the discovery procedure will stay active until a P2P group is formed or a successful connection request is initiated [1]. When the application knows that peer devices are discovered (this is described later in this section), it can request for the current list of devices from the framework by calling the *requestPeers* method. Figure 2.9 shows a proposed state machine of the discovery process with a suggested sequence of the related request methods. The WIFI_P2P_PEERS_CHANGED_ACTION intent action will be explained later in this section.

When the current list of peer devices has been received and the application enters

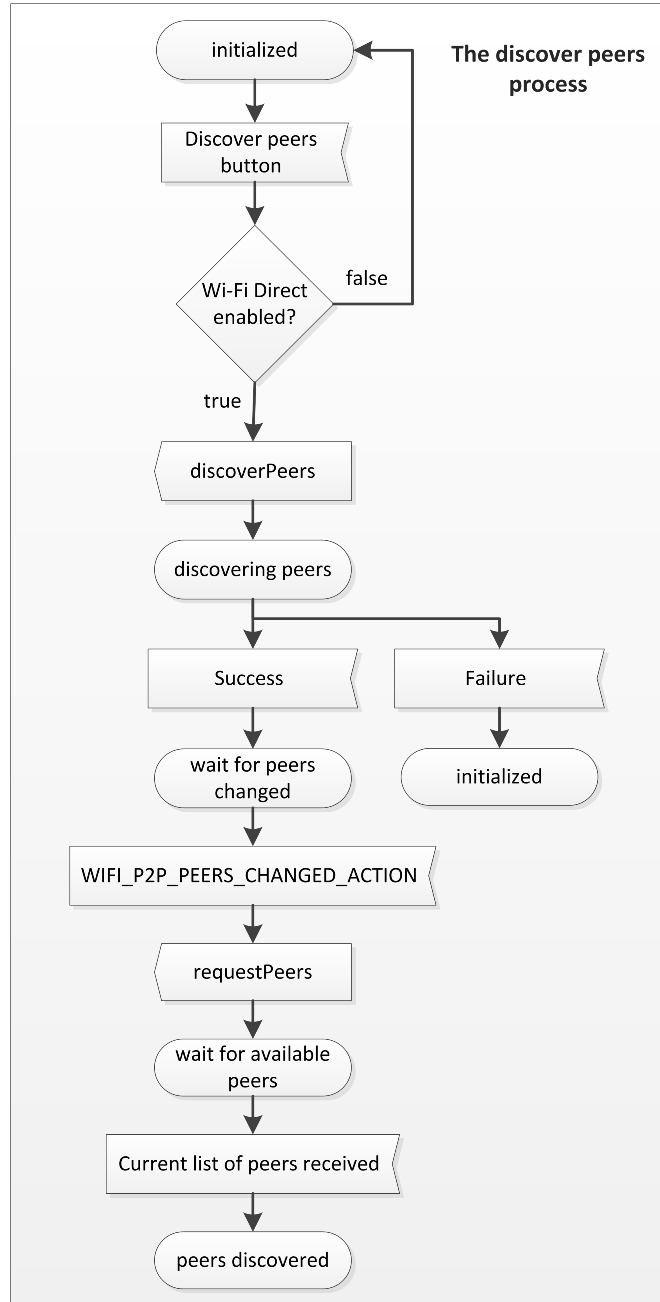


Figure 2.9: A state machine of the discovery process

the *peers discovered* state in Fig. 2.9 and Fig. 2.10, a connection request can be initiated with one of the devices in the list. This is done by executing the *connect* method. If the current device is not already part of a P2P group, this request will initiate a group negotiation with the peer device [1]. A group negotiation is required in order to decide which device that is going to act as the group owner. If the current device is already part of a group, an invitation to join this group is sent. If an ongoing group negotiation ought to be cancelled, the *cancelConnect* method must be executed. Upon a successful group negotiation and when the

application knows that the connection has been changed (this is described later in this section), it can detect if network connectivity exists. If so, a request for connection info can be inquired by executing the *requestConnectionInfo* method. By doing so, the application will be able to attain the following details:

- If a group has been formed.
- The group owner's IP address.
- If the current device is the group owner.

If a group has been formed, a request for group info can be inquired by executing the *requestGroupInfo* method. The following information will be received by the application:

- The list of client devices that are currently part of the P2P group.
- The name of the interface the group is using (e.g. *p2p-wlan0-0*).
- The Service Set Identifier (SSID) of the group (e.g. *DIRECT-fd*).
- The details of the group owner in a *WifiP2pDevice* object.
- The group's passphrase.
- If the current device is the group owner.

Figure 2.10 shows a proposed state machine of the connection process with a suggested sequence of the relevant request methods.

The *createGroup* method causes the current device to create an empty P2P group with itself acting as the group owner. This method is only intended to be used in circumstances where the peer devices are legacy equipment, and will normally not be used in ordinary Wi-Fi Direct operations. Nevertheless, this request method is particularly useful nowadays since the Wi-Fi Direct technology is still pretty novel to most consumers' equipment.

In order to perform a disconnection request to a connected group, the *removeGroup* method must be executed. By using the method's callback listener, the application will be able to know whether the request was successful or not. Figure 2.11 shows a proposed state machine of the disconnection process. This diagram shows that the removeGroup method should be able of being triggered in the following states:

- *wait for connection changed*

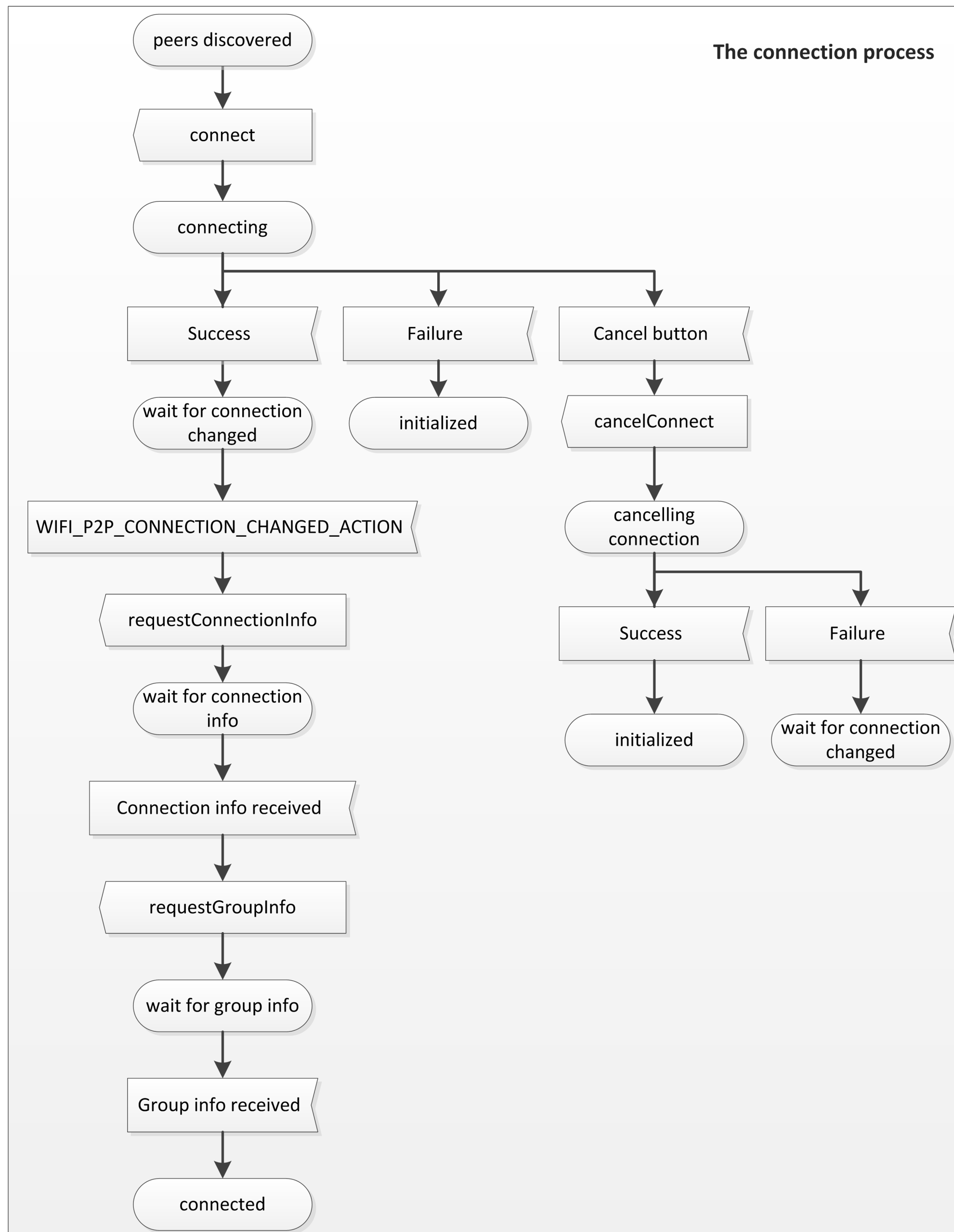


Figure 2.10: A state machine of the connection process

- *wait for connection info*
- *wait for group info*
- *connected*

Intents and intent actions

In order for a Wi-Fi Direct application to be able to know when certain Wi-Fi Direct specific events happen, it needs to listen for broadcast intents. To achieve

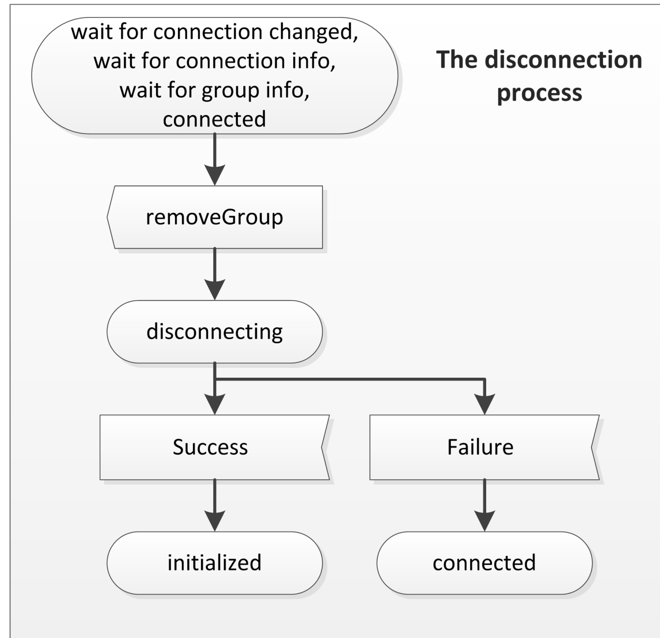


Figure 2.11: A state machine of the disconnection process

this, broadcast receivers with intent filters need to be registered at the application. The following intent actions are relevant for Wi-Fi Direct [1]:

- *WIFI_P2P_STATE_CHANGED_ACTION*
- *WIFI_P2P_PEERS_CHANGED_ACTION*
- *WIFI_P2P_CONNECTION_CHANGED_ACTION*
- *WIFI_P2P_THIS_DEVICE_CHANGED_ACTION*

To be able to initiate a discovery process and connect with peer devices, the Wi-Fi Direct mode must be enabled on the current device. This information will be broadcasted by the Android system. By registering a broadcast receiver with an intent filter that contains the WIFI_P2P_STATE_CHANGED_ACTION intent action, the application can be informed if the Wi-Fi Direct mode is enabled on the current device. Every time the users turn on or off the Wi-Fi Direct mode, an intent will be broadcasted causing the application to always be updated on the Wi-Fi Direct mode's status.

If the WIFI_P2P_PEERS_CHANGED_ACTION intent action is added to the filter, the application can be notified when peers are discovered. This will most likely occur after a peer discovery process has been initiated. However, the application will always be ready to receive this type of notification as long as the broadcast receiver is registered.

In order for the application to detect any changes in its Wi-Fi connectivity, the WIFI_P2P_CONNECTION_CHANGED_ACTION action must be added to the filter. There are several reasons for this intent action to be triggered, e.g.:

- A connection request is made by the current device and the group negotiation procedure was successfully accomplished.
- The connection was lost.
- A disconnection procedure was successfully accomplished.
- Another device has sent a connection request to the current device and the group negotiation procedure was successfully accomplished.

By adding the WIFI_P2P_THIS_DEVICE_CHANGED_ACTION intent to the filter, the application will be informed when a change in the device's P2P properties has occurred. This could for instance happen when the devices status changes from being available to be connected. When the intent action is triggered, the UI of the application should be updated in order to give back information of the device's current status to the users.

Permissions and uses-features

Permissions provide a restricted access to part of the device's code or to some specific data on the device [14]. The reason for this is to protect critical code or data from being misused. They are identified by a unique label and the users accept the application's permissions prior to installing it.

By using the API of Wi-Fi Direct, the application must request the following permissions[15]:

- *ACCESS_WIFI_STATE*
- *CHANGE_WIFI_STATE*
- *INTERNET*

In order to let the application access information of the Wi-Fi networks the ACCESS_WIFI_STATE permission must be authorized [16]. In addition, in order to change the Wi-Fi connectivity, the CHANGE_WIFI_STATE permission must be authorized. Even though an internet connection is not required for a Wi-Fi Direct application, the INTERNET permission must be authorized. This permission authorizes the application to open Java network sockets, which is necessary when

communicating to other peer devices over Wi-Fi Direct.

Uses-features are announcements of specific hardware and software the application are using [17]. They are only meant to inform external entities, which means that the Android system does not check whether the device actually support these features or not. Google Play uses for instance these declarations to filter applications that are visible to the users. This is done by comparing the Google Play applications' uses-features with the features available on the device.

Since only a fraction of Android devices are Wi-Fi Direct compatible, the *android.hardware.wifi.direct* uses-feature should be declared.

Chapter 3

Methodology

In this chapter, a discussion on which method we opted to follow is carried out. Furthermore, a description of how we preceded the development of the building blocks, based on the chosen method is presented. In addition, an issue we encountered during the testing procedure is discussed. This issue is worth mentioning because it affected the testing process, and it persisted during the entire development process. Finally, a short description of which tools we used in order to obtain the objective is presented.

3.1 The choice of method

The building blocks and the ESMs of Arctis SDK are based on the UML 2.0 semantics [18]. UML is a modeling language standardized by the Object Management Group (OMG) for describing a software system in a family of graphical notations [19, 20]. This method is known as Model-Driven Engineering (MDE).

Even though models play an important part of the Arctis tool, it may be unsuitable to classify the method of using Arctis as MDE. As France and Rumpe [21] state, «The term Model-Driven Engineering (MDE) is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations.» This means that developers need to have a distinct separation between design and implementation. To be precise, developers need to be finished with the design of the models before they are implemented. This method of development follows the *waterfall model*, where the different activities are proceeded sequentially through various phases [22] (see Fig. 3.1 for an illustration). Every activity that belongs to a phase must

be completely finished before the next phase starts [23]. Since there are many uncertainties in the design phase, this method is quite difficult to follow.

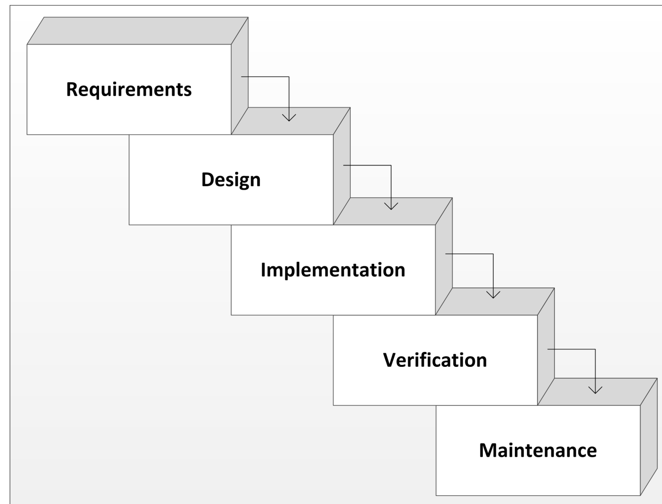


Figure 3.1: The waterfall model. Adapted from [24]

It is often necessary to revert from the implementation phase and redesign already predefined models, because of some different points of views has emerged. Therefore, we based our methodology on the *iterative model* (see Fig. 3.2 for an illustration). By using the Arctis SDK, we were able to continuously alternate between model design and code implementation through testing and evaluation. This process is exemplified in Sect. 5.5.

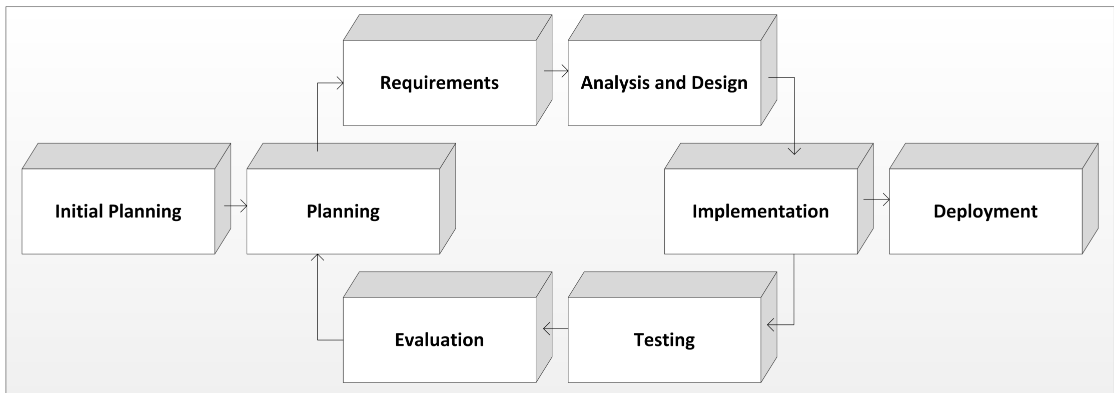


Figure 3.2: The iterative model. Adapted from [24]

3.2 The development process

Figure 3.3 shows an overview of how we managed to reach the objective of developing Arctis building blocks to support Wi-Fi Direct. Since the Wi-Fi Direct technology was unfamiliar to us, gaining sufficient knowledge within this area was the first phase of the work. By obtaining and studying an adequate amount of

Figure 3.3: An overview of the development process

background material, an investigation of Android's Wi-Fi Direct API was initiated. By doing so, we realized that the API documentation was difficult to follow with asynchronous messages being sent from different sources. Hence, we designed some informal state machines based on the SDL semantic in order to enhance our own comprehension of the API. During this we realized that we could divide the API's functioning into the following four operational procedures:

- Initialization
- Discovering of peers
- Connection
- Disconnection

With a broader understanding of the Wi-Fi Direct API and the Wi-Fi Direct technology, a dialog with some employees from Pixavi was commenced. During this process, we discussed various scenarios and use-cases on how to implement Wi-Fi Direct into theirs system. This is documented in Chapt. 4. The benefit of having Pixavi involved was to get some genuine experience on how such technology should be implemented in order to meet the requirements of usability. When an agreement on some specific use-cases was established, informal sequence diagrams was created in order to gain an unambiguous foundation for the further developments of the API blocks.

Since our objective was to prove that predefined building blocks based on the Wi-Fi Direct API could easily be implemented in other developer's applications, we needed a foundation to run experiments and tests. Hence, we opted to create an example application assembled by these blocks. As a result, these blocks became composed by having an iterative process between testing and restructuring.

In addition to run various tests on the blocks, we needed to analyze them. This was done both manually and automatically. Arctis let us set up ESMs in conjunction with the blocks to control the sequence of the various messages that is sent and received. By doing so, we were able to have them behave in accordance with the predefined state machines and sequence diagrams. In order to analyze the behavior, Arctis allows us to animate a token flow by manually stepping through the ESM. While doing this, Arctis automatically checks for design flaws and reports back if something was found.

In order for a block to comply with our objective, it needs to enhance the simplicity for a developer of having a functionality implemented. In addition, it must be generic enough to suite most relevant tasks. This was discussed in Sect. 1.1. Finally after multiple iterations of testing, analysis and reshaping, the building blocks were satisfactory to meet our requirements of functionality and simplicity. This resulted in a complete set of Wi-Fi Direct API building blocks endorsed by the example application.

3.3 Connection issue

During the testing procedure, we observed some unexpected erroneous events. After four successive times of connection and disconnection between two devices, they suddenly wouldn't reconnect. However if they were rebooted, everything went back to normal and worked as it should, until the process was repeated. This error

has a significant impact on the user-friendliness and should not be acceptable. Thus, we needed to investigate this in order to detect if the implementation or the design choices was the source of the error. By logging the events using *logcat* [25], we recognized that every time a connection is established the name of the interface was changed in the following way:

- *p2p-wlan0-0*
- *p2p-wlan0-1*

The last number continued to increment each time a new connection was established until a connection request was sent for the fifth time. Then the following message was logged:

- *Failed to create interface p2p-wlan0-4: -12 (Out of memory)*

From this log we could presume that every time a new connection was initiated, a virtual interface was established and the last number identifies this virtual interface. However, when a disconnection is performed it looks like the virtual interface is not properly removed. Hence, this could be the reason why the last number is incremented on connections subsequent to disconnections. By searching the web for the issue, we found several discussion forums where this error has been confirmed. Therefore, we concluded that our design and implementation was not related to this issue, and it was located at a lower layer in Android's software stack. This error will most likely be corrected in the future, and the correction will probably not affect our design since it is based on the Wi-Fi Direct API.

3.4 Development environment

All development was realized in Eclipse Classic 3.7.2 (Indigo) running on the Windows 7 operating system. The building blocks were constructed using Arctis plug-in 1.0.0.M0642 for Eclipse, and the Android specific features was provided by the Android Development Toolkit 16.0.1. The testing was performed on two Samsung Galaxy Nexus GT-I9250 smartphones running on Android 4.0.2 (Ice Cream Sandwich) operating system.

Chapter 4

System Implementation Design

This chapter is confidential, and therefore omitted in this version of the thesis.

Chapter 5

API Building Blocks

Design of reusable building blocks for Wi-Fi Direct was achieved by the use of Arctis SDK. The objective was to end up with blocks that are easily implementable in other applications. As a result, the requirement of comprehending the technological details of Wi-Fi Direct and Android's Wi-Fi Direct API would be relaxed. Hence, these blocks were made generic in such a way that they can be used independent of application's use-case. In this manner, the restriction of functionality relies in how the blocks are combined, not the blocks itself.

This chapter describes these building blocks in details, as well as how they are composed to structure the complete functionality. The most important functions of Wi-Fi Direct are the following:

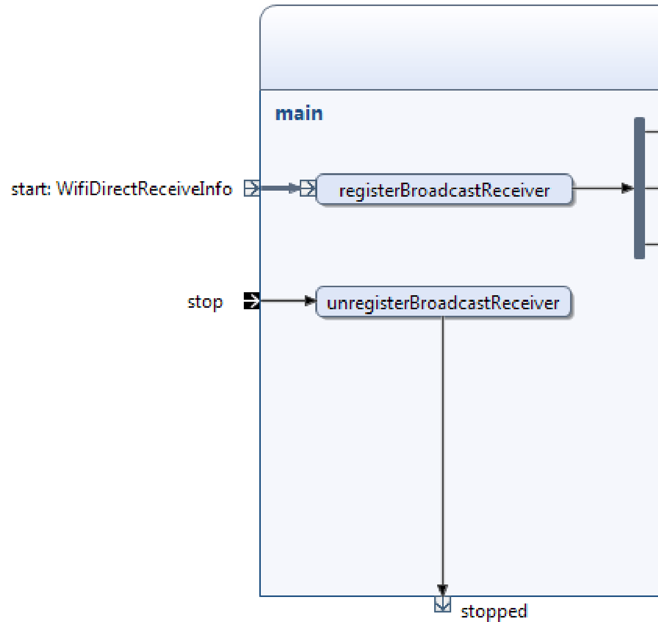
- Be able to find peer devices in range.
- Send connection requests.
- Listen for connection requests by peer devices.

These functions are separated into three different building blocks given the following names:

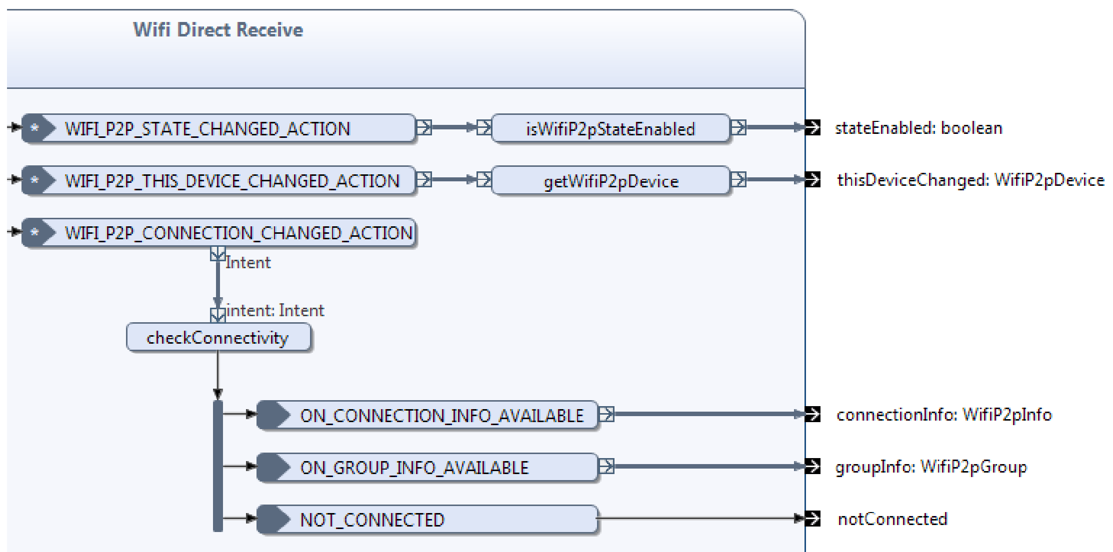
- *Wifi Direct Receive*
- *Wifi Direct Discover Peers*
- *Wifi Direct Connect*

5.1 The Wifi Direct Receive block

This block is responsible for notifying the application when some Wi-Fi Direct related intents are sent from the Android framework. See Fig. 5.1 for the block's internal structure.



(a) The left side of the block



(b) The right side of the block

Figure 5.1: The internal structure of the Wifi Direct Receive block

5.1.1 Description of the Wifi Direct Receive block

This block is initiated when the *start* pin gets triggered (see Fig. 5.1a). This pin has a data type of the *WifiDirectReceiveInfo* class. This class contains two public variables of the data type *WifiP2pManager* and *Channel*. The reason for using this data type is that these variables are declared outside of the block, but is also needed here.

Within the *registerBroadcastReceiver* operation, two private variables are initialized and a broadcast receiver is registered. This broadcast receiver has filtered out the following intent actions (see Sect. 2.3.2 for a detailed description of these intent actions):

- WIFI_P2P_STATE_CHANGED_ACTION
- WIFI_P2P_THIS_DEVICE_CHANGED_ACTION
- WIFI_P2P_CONNECTION_CHANGED_ACTION

The *stateEnabled* pin will return a boolean variable depending on the received intent's Wi-Fi P2P state (see Fig. 5.1b). If the state is enabled, which means that the Wi-Fi Direct mode is turned on, a true variable is returned. Otherwise a false variable will be returned. This test is done in the *isWifiP2pStateEnabled* operation.

The *thisDeviceChanged* pin will return P2P related information of the current device. This happens when a P2P related event has changed something with the device. The event could for instance result in a change in the device's status. The device's variable of data type *WifiP2pDevice* is extracted from the intent in the *getWifiP2pDevice* operation.

If the Wi-Fi Direct framework notifies that the device's connectivity has been changed, an intent will be sent. Consequently, the broadcast receiver will receive this intent and the block will check its connectivity in the *checkConnectivity* operation. If the device has a connection, the *connectionInfo* pin returns the Wi-Fi Direct connection information. If a group is formed, the *groupInfo* pin returns the Wi-Fi Direct group information. Otherwise, if the device doesn't have a connection, the *notConnected* pin will be triggered.

To unregister the broadcast receiver, the stop pin must be triggered. This will in addition result in a termination of the building block. The deregistration happens in the *unregisterBroadcastReceiver* operation.

5.1.2 Analysis of the Wifi Direct Receive block

Figure 5.2 shows the Wifi Direct Receive block's ESM. It has two states in addition to the initial and the final state, namely *active* and *stopping*.

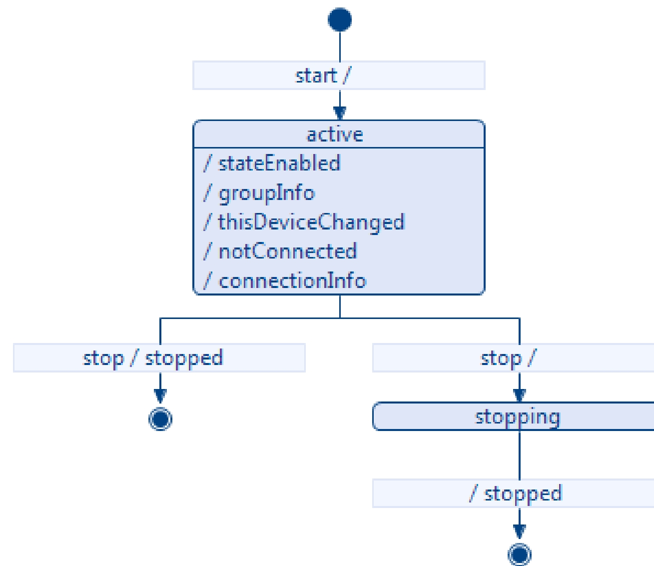


Figure 5.2: The ESM of the Wifi Direct Receive block

When the start pin is triggered, the block enters the active state and is able to trigger the five different output pins shown under this state in Fig. 5.2. In addition to these output pins, the stop pin can be triggered. This is the only event that will cause the block to leave the active state. If this happens, the block will enter stopping state and wait for the stopped pin to be triggered, in order to enter the final state. Otherwise, the stopped pin could instantly be triggered and the block enters the final state.

By deciding the order of pins to be triggered, the block will in this case not be able to trigger any output pins when it is in its initial state. It will correspondingly only be terminated when it is in the active or stopping state.

5.2 The Wifi Direct Discover Peers block

This block is responsible for finding peer devices and to return the current list of peer devices found. See Fig. 5.3 and Fig. 5.4 for the internal structure of the Wifi Direct Discover Peers block.

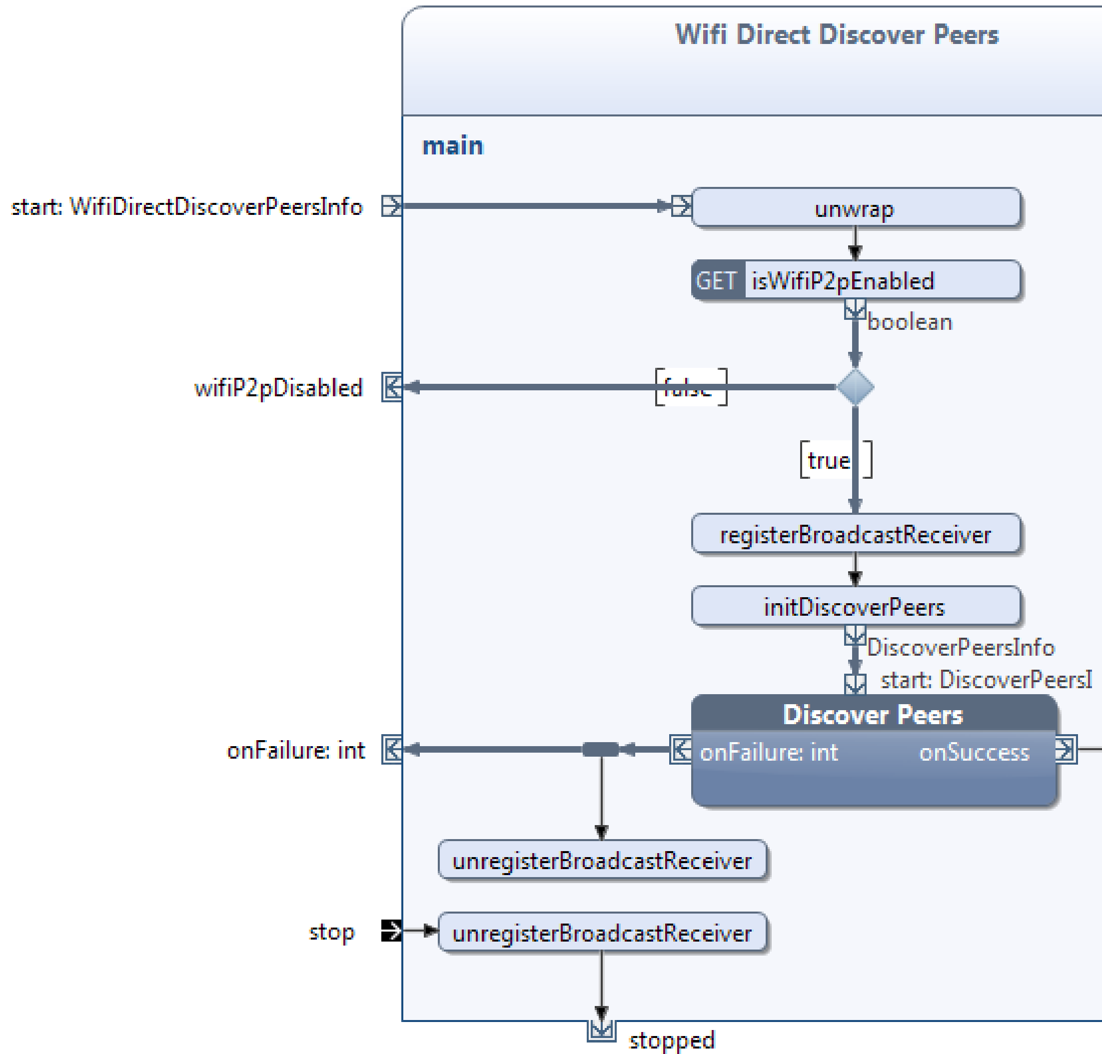


Figure 5.3: The left side of the Wifi Direct Discover Peers block

5.2.1 Description of the Wifi Direct Discover Peers block

In the same way as with the Wifi Direct Receive block, this block is initiated when start pin is triggered (see Fig. 5.3). This pin contains a data type of the *WifiDirectDiscoverPeersInfo* class. This class contains three public variables of the following data types:

- *Channel*
- *boolean*
- *WifiP2pManager*

These variables will initialize some private variables that belongs to the block's class in the *unwrap* operation. The boolean data type is supposed to be denoted

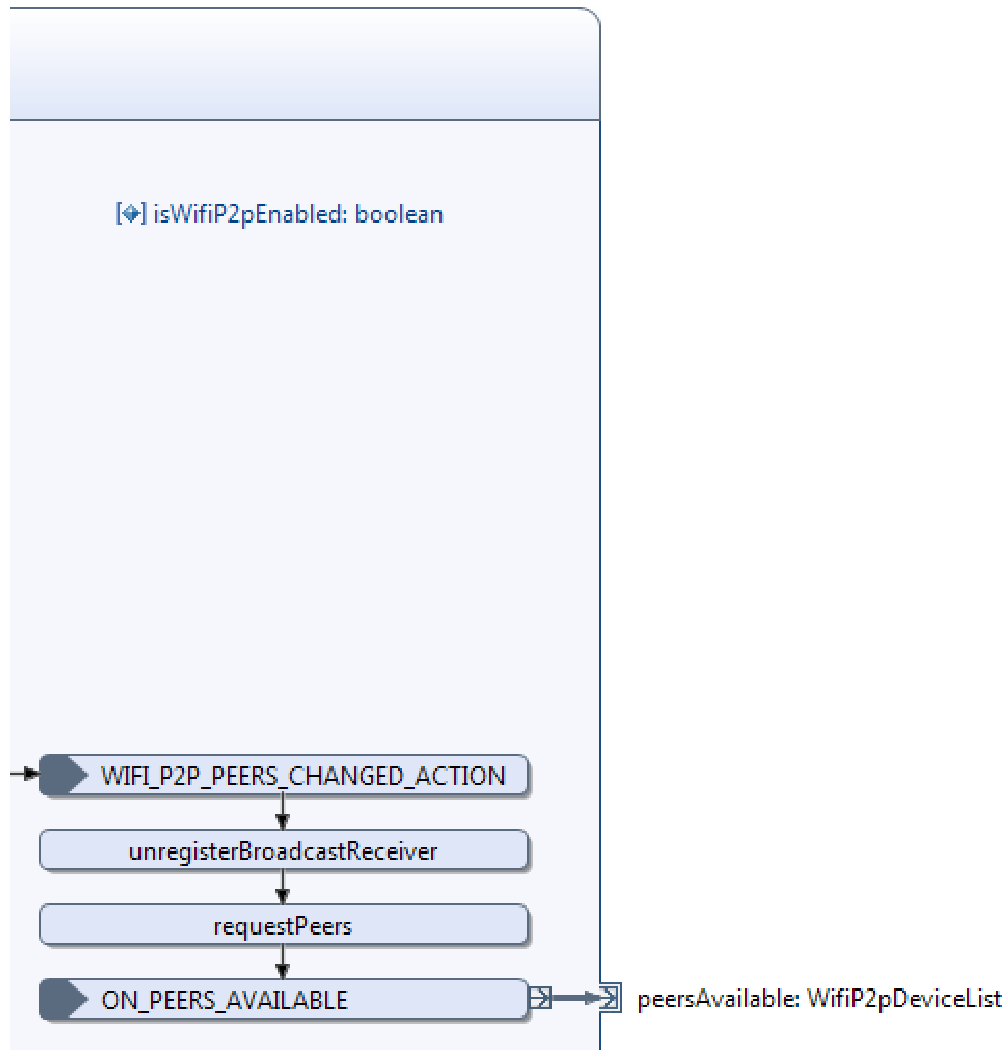


Figure 5.4: The right side of the Wifi Direct Discover Peers block

by the stateEnabled pin in the Wifi Direct Receive block. This means again that if this variable is false, the Wi-Fi Direct mode is turned off. Consequently, the *wifiP2pDisabled* pin is triggered and the block terminates. Otherwise, a broadcast receiver is registered in the *registerBroadcastReceiver* operation. In addition, the Discover Peers block is prepared to be initiated in the *initDiscoverPeers* operation. The details of these operations are described in Sect. 1.2.

The *Discover Peers* block shown in Fig. 5.5 contains only one operation identified as *discoverPeers*. This operation is directly associated with the discoverPeers request method in the API (see Sect. 2.3.2 for more details). The block will immediately be notified whether the request was successful or a failure. Upon receiving this notification, the block will terminate by either trigger the *onSuccess* or the *onFailure* pin depending on the response. The onFailure pin includes a data type

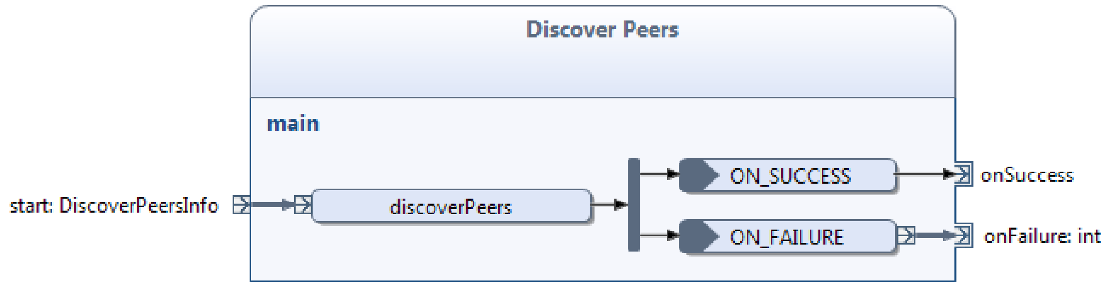


Figure 5.5: The internal structure of the Discover Peers block

of *int* which represents the error code.

If the Discover Peers block triggers the onFailure pin, the previously registered broadcast receiver is deregistered in the unregisterBroadcastReceiver operation (see Fig. 5.3). In addition, the onFailure pin is triggered and the Wifi Discover Peers block gets terminated.

If the onSuccess pin is triggered, an event will cause the block to wait for an intent to be sent from the Android framework. This and the following procedures are described in details in Sect. 1.2. When the *peersAvailable* pin is triggered, a data type of *WifiP2pDeviceList*, which essentially is a list of peer devices is conveyed out of the block.

To manually deregister the broadcast receiver, the stop pin must be triggered. This is done in the same matter as with the Wifi Direct Receive block.

5.2.2 Analysis of the Wifi Discover Peers block

Figure 5.6 shows the Wifi Discover Peers block's ESM. It has one state identified as *discoveringPeers*, in addition to the initial and final state.

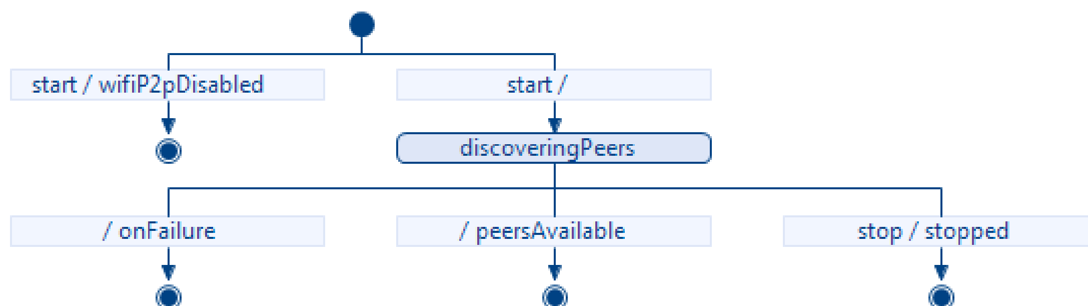


Figure 5.6: The ESM of the Wifi Discover Peers block

When the block is in its initial state, two events can happen. The start pin could either trigger the block to enter the final state or the discoveringPeers state. In case

the block enters the final state, the wifiP2pDisabled pin is triggered. Otherwise, if the block enters the disoveringPeers state, three events can happen where all of them cause the block to enter the final state:

- The onFailure pin is triggered.
- The peersAvailable pin is triggered.
- The stop pin is triggered, resulting the stopped pin to be triggered.

5.3 The Wifi Direct Connect block

This block is responsible for connecting to a peer device in addition to tear down a connection. See Fig. 5.7 and Fig. 5.8 for the block's internal structure.

5.3.1 Description of the Wifi Direct Connect block

This block is initiated when the *connect* pin is triggered. In the same way as with the Wifi Direct Receive and the Wifi Direct Discover Peers block, this block uses variables that are declared outside the block. Hence, the connect pin includes the data type of the *WifiDirectConnectInfo* class. This class has the three public variables of the following data types:

- *Channel*
- *WifiP2pConfig*
- *WifiP2pManager*

In the *setParameters* operation, these variables are initialized to the following classes' public variables:

- *ContactInfo*
- *CancelConnectInfo*
- *RemoveGroupInfo*

These classes are used in the *Connect*, the *Cancel Connect* and the *Remove Group* block in the same way as how the Discover Peers block uses the DiscoverPeersInfo class in the Wifi Direct Discover Peers block. These blocks are equally constructed, with one start, one onSuccess and one onFailure pin each. The only difference between them lies in the operations, which contains a distinct request method for each of the blocks. See Fig. 5.9 for the internal structure of these blocks.

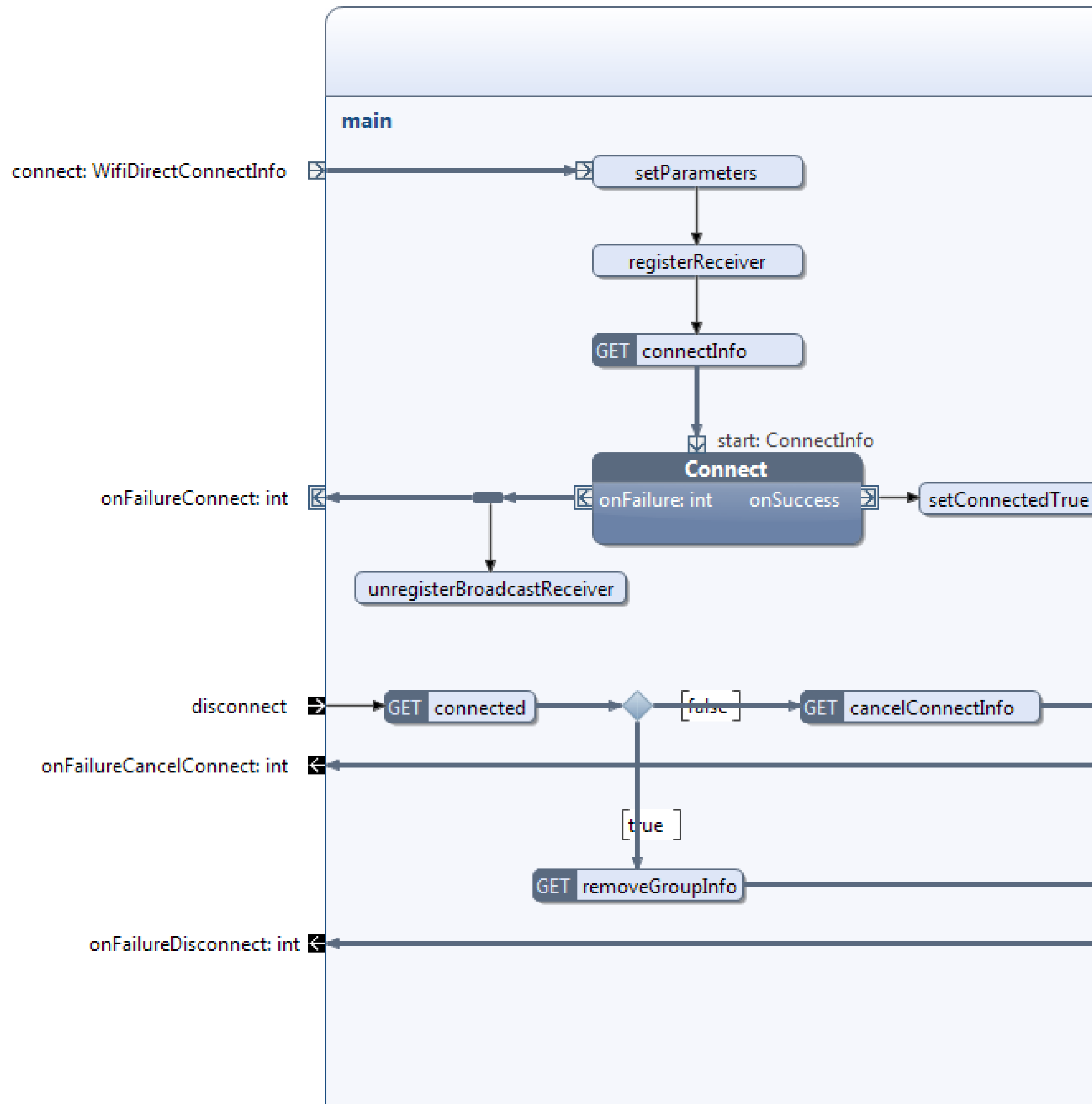


Figure 5.7: The left side of the Wifi Direct Connect block

As with the Wifi Direct Receive and the Wifi Direct Discover Peers, this block uses a broadcast receiver. This receiver is registered in the *registerReceiver* operation with a WIFI_P2P_CONNECTION_CHANGED_ACTION intent action in its intent filter.

The Connect block starts a P2P connection to a peer device with a specified configuration. This configuration is given by the WifiP2pConfig data type, which is configured outside of the Wifi Direct Connect block.

If the Connect block triggers the onFailure pin, the previously registered broadcast receiver will be unregistered and the *onFailureConnect* pin is triggered (see Fig. 5.7). Consequently, the Wifi Direct Connect block terminates. If the Connect

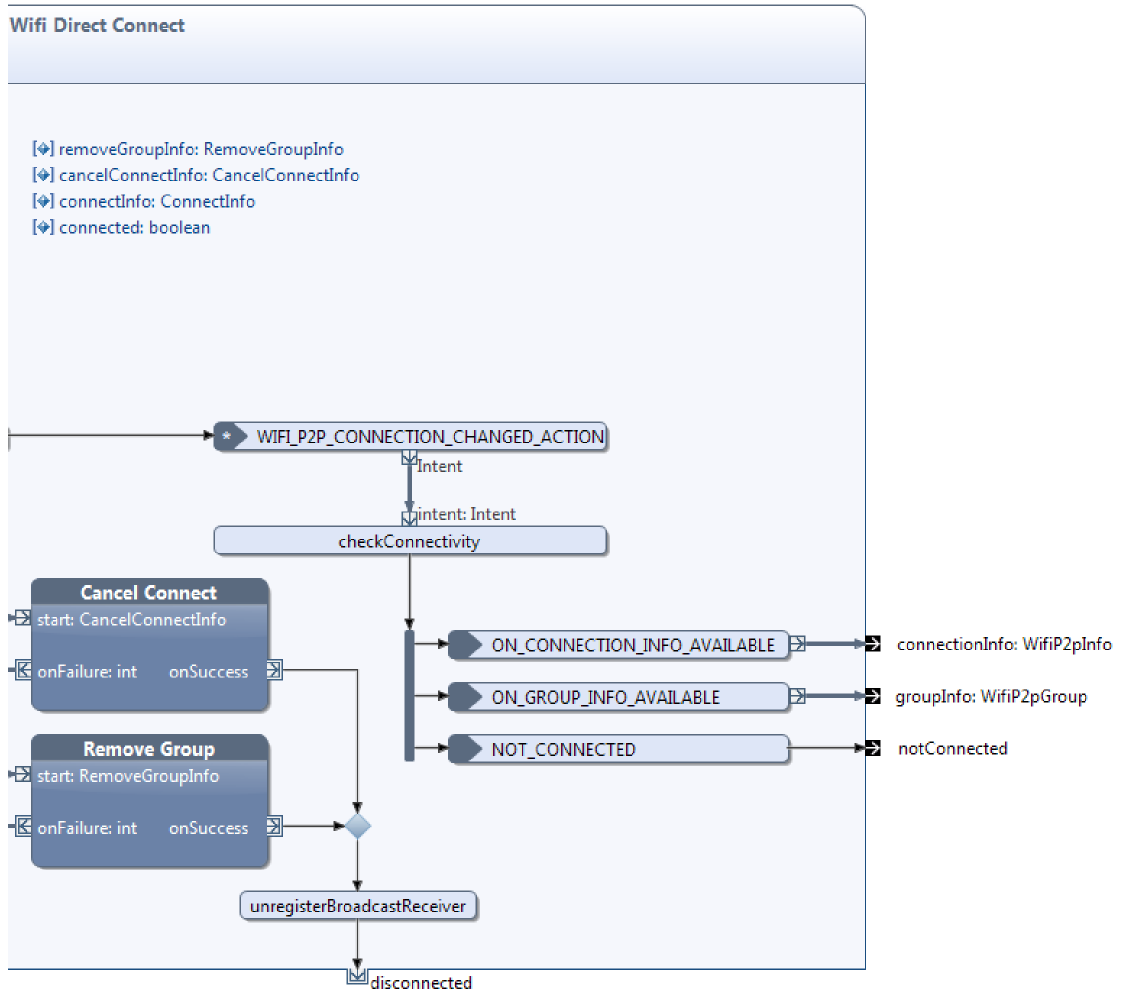
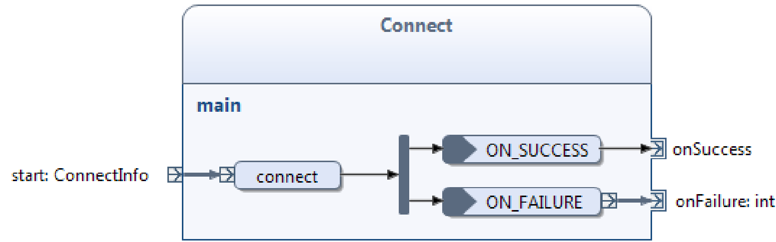


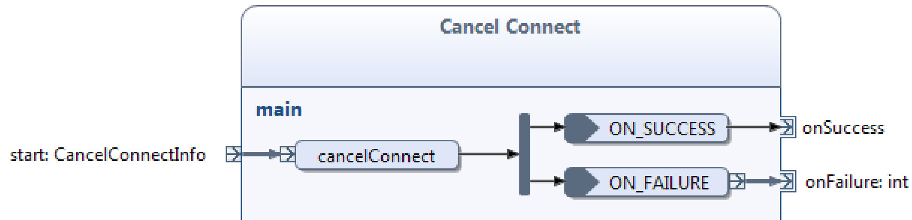
Figure 5.8: The right side of the Wifi Direct Connect block

block triggers the onSuccess pin, a boolean variable is declared true in the *setConnectedTrue* operation. This variable indicates whether the connection request was successful or not and is needed when a disconnection request is performed. The block will then wait for an intent to be received by the previously registered broadcast receiver. When the intent is received, the connectivity is checked in *checkConnectivity* operation. This and the following procedures are exactly the same as with the checkConnectivity operation and its subsequent procedures in the Wifi Direct Receive block.

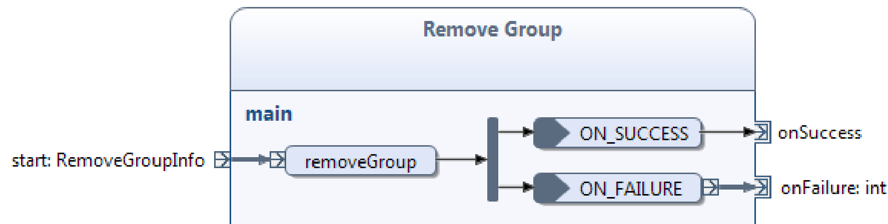
If the users want to disconnect or abort an ongoing P2P group negotiation, the disconnect pin must be triggered. Consequently, either the Cancel Connect block or the Remove Group block will be initiated. Which of them are determined by the boolean variable mentioned in the previous paragraph. A successful connection request means that a group negotiation has been successful. This further means that a group has been formed. In this case, an abortion of the group negotiation



(a) The Connect block



(b) The Cancel Connect block



(c) The Remove Group block

Figure 5.9: The internal structure of the Connect, Cancel Connect and Remove Group block

is too late to perform and the Remove Group block should be initiated instead of the Cancel Connect block. If the disconnect pin is triggered prior to a successful connection request, the boolean variable has its default value, which is false. This will lead to the initiation of the Cancel Connect block instead. As a result, a termination request to the ongoing group negotiation is executed.

If either the Cancel Connect or the Remove Group block triggers theirs onFailure pin, the corresponding *onFailureCancelConnect* or *onFailureDisconnect* pin gets triggered. Accordingly, the failure code will be transported out of the Wifi Direct Connect block. Otherwise, if the procedure is successful and the onSuccess pin is triggered at one of these blocks, the previously registered broadcast receiver gets unregistered. Subsequently, the *disconnected* pin at the Wifi Direct Connect block is triggered and the block terminates.

5.3.2 Analysis of the Wifi Direct Connect block

Figure 5.10 shows the Wifi Direct Connect block's ESM. It has three states besides the initial and final state, namely *connecting*, *connected* and *disconnecting*.

When the connect pin initiates the block, it will enter the connecting state. In this state, five different pins are able to trigger the block to enter three different states:

- onFailureConnect cause the block to enter the final state.
- connectionInfo, notConnected and groupInfo cause the block to enter the connected state.
- disconnect cause the block to enter the disconnecting state.

Since the users should be able to abort an ongoing group negotiation, the disconnect pin must be enabled in the connecting state. When the block is in the connected state, the groupInfo and connectionInfo pins are enabled to be capable of continuously sending updated group and connection information out of the block.

The only pin that is able to trigger the block out of the connected state is the disconnect pin. When this happens, the block enters the disconnecting state while it tries to tear down the connection. As pointed out earlier, this operation can either be successful or a failure. If it fails, either the onFailureDisconnect or the onFailureCancelConnect pin is triggered depending on which of the Cancel Connect and Remove Group block that was initiated. If the onFailureDisconnect pin was triggered, the block returns to the connected state. Otherwise, if the onFailureCancelConnect pin was triggered the block will return to the connecting state. If the disconnection procedure succeeded, the disconnected pin gets triggered and the block enters the final state causing it to terminate.

5.4 Combining the API blocks

All blocks defined so far make up the total Wi-Fi Direct service. However, combining these blocks to establish the total service still requires considerable background knowledge of Android's Wi-Fi Direct API and the Wi-Fi Direct technology. As a consequence, we designed a block named *Wifi Direct Service* to encapsulate these blocks.

A consequence of reducing the complexity is that the functionality also becomes reduced. To be precise, this block automatically initiates connection request to

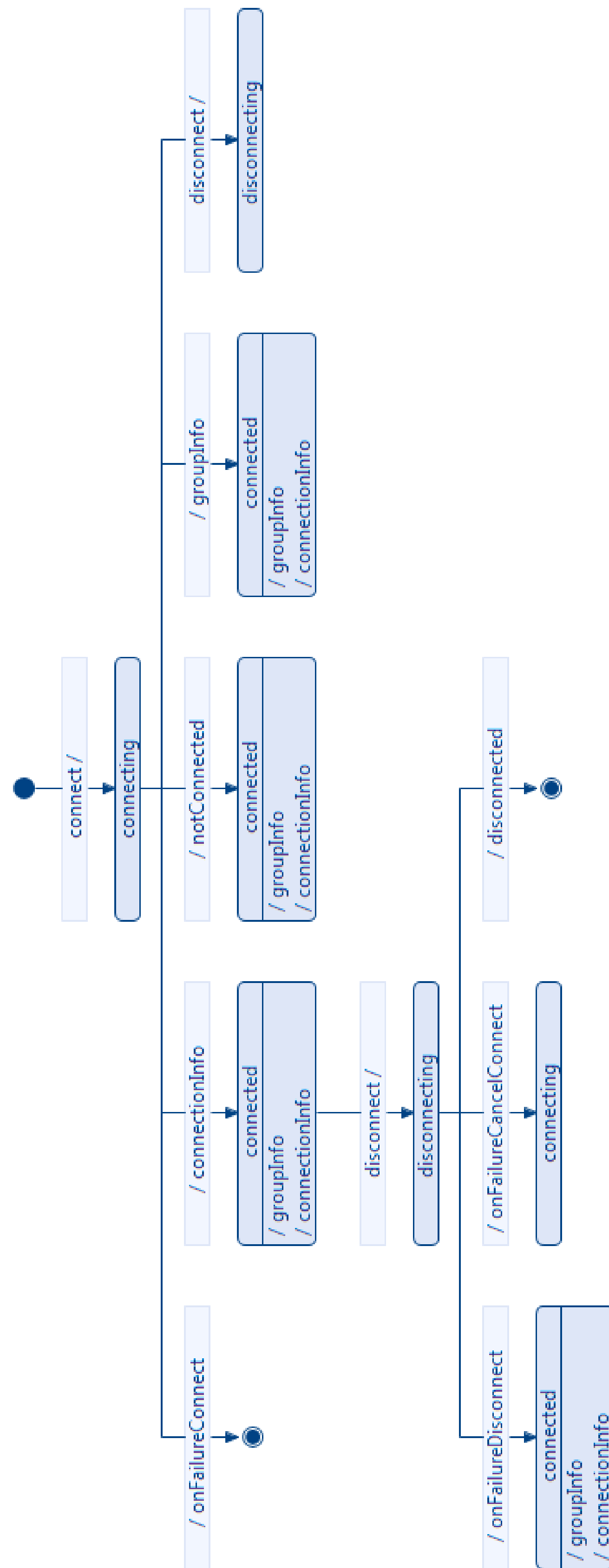


Figure 5.10: The ESM of the Wifi Direct Connect block

peer devices that are discovered instead of letting the user decide when this should happen. If this assumption is satisfactory, the block will be usable. Otherwise, other blocks could be constructed to suite other use-cases. See Fig. 5.11, Fig. 5.12 and Fig. 5.13 for this block's internal structure.

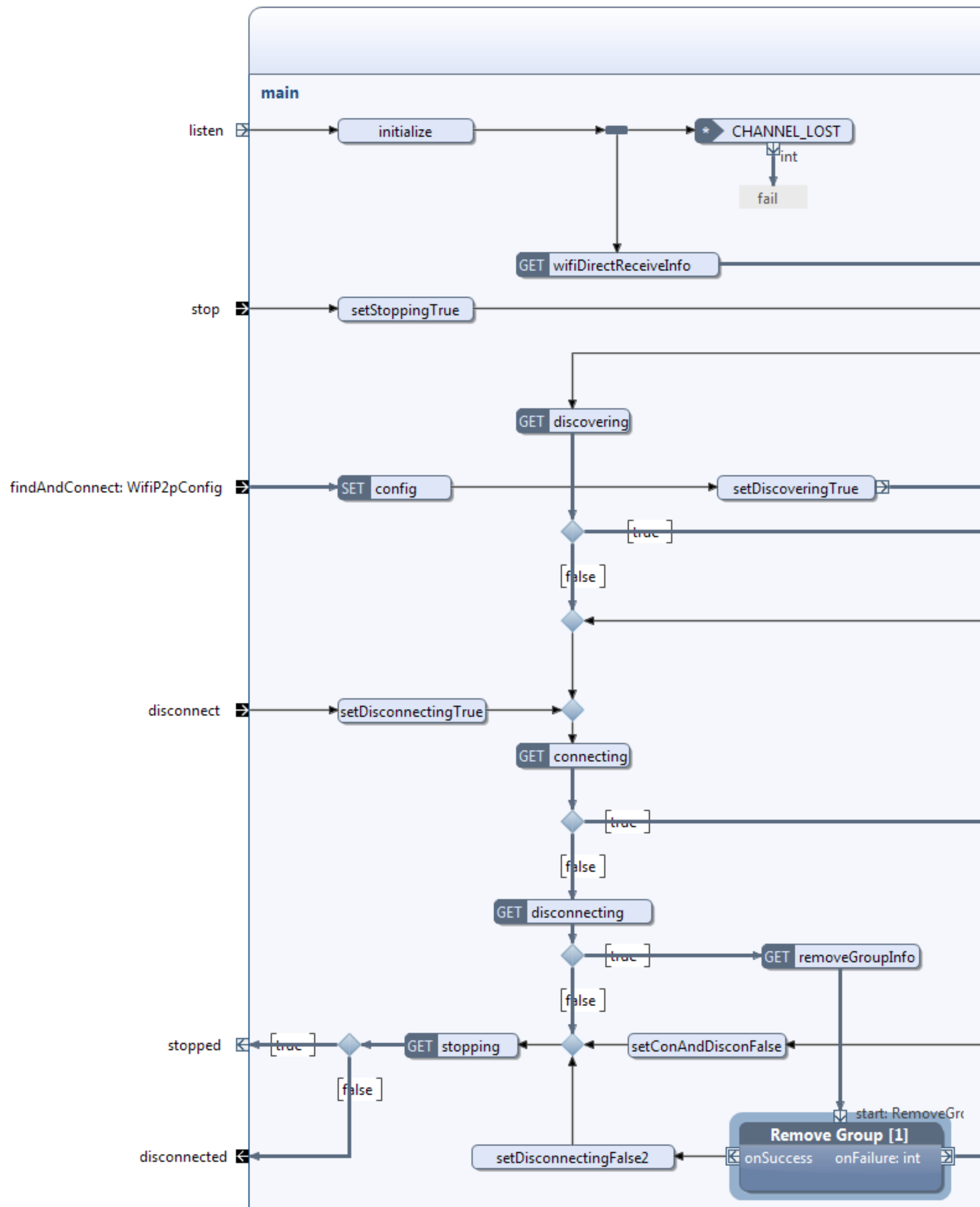


Figure 5.11: The left side of the Wifi Direct Service block

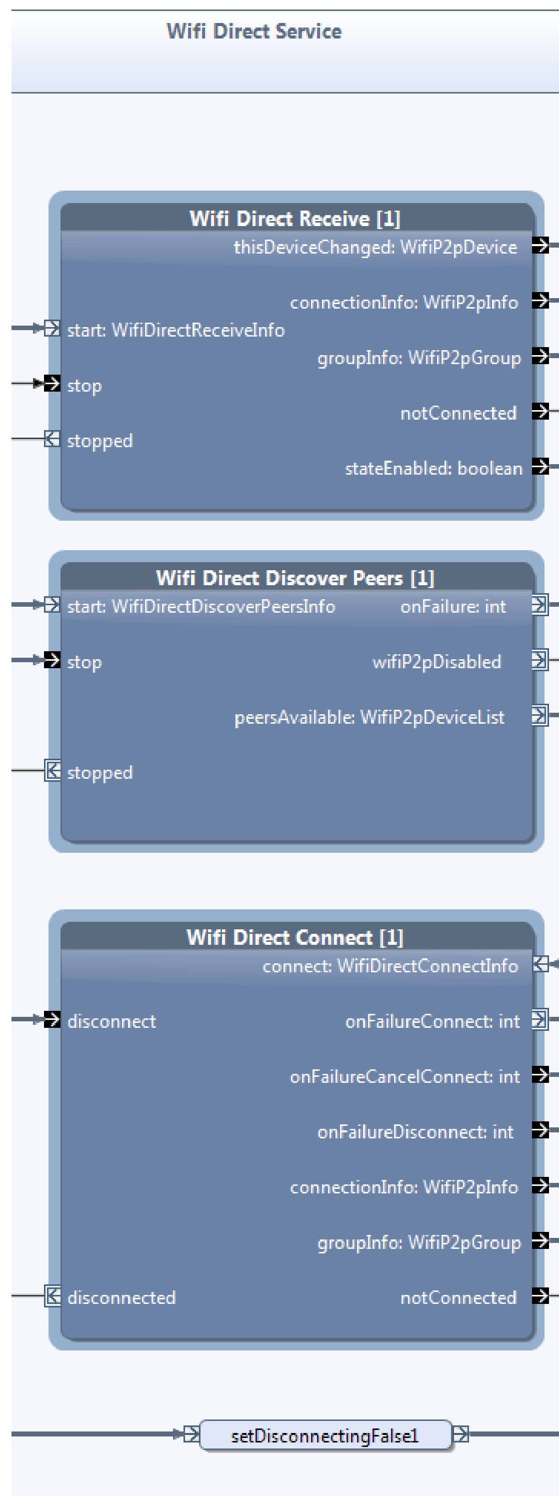


Figure 5.12: The middle part of the Wifi Direct Service block

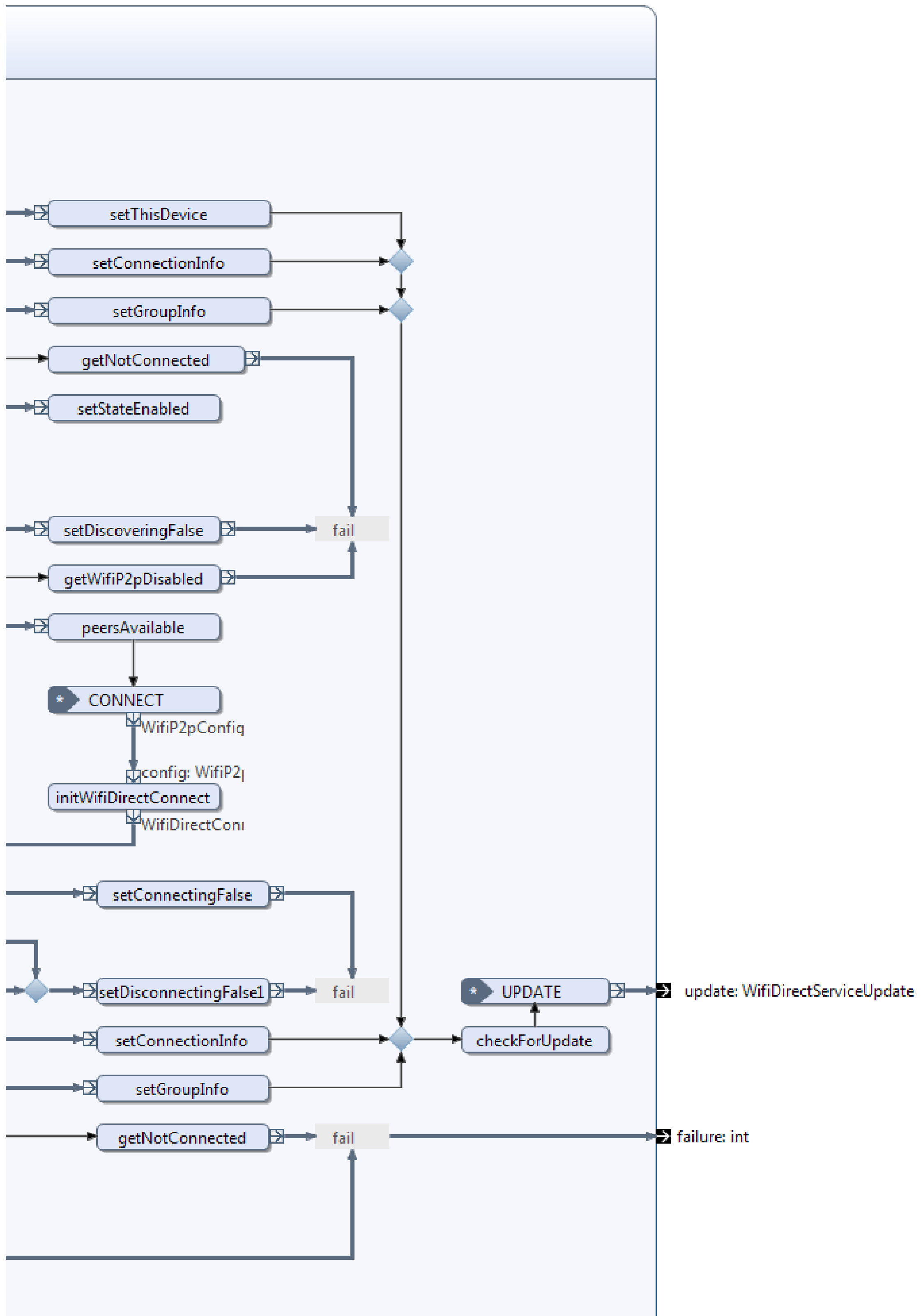


Figure 5.13: The right side of the Wifi Direct Service block

5.4.1 Description of the Wifi Direct Service block

This block is initiated when the *listen* pin is triggered. This will cause the *initialize* operation to be activated. Within this operation, two variables of the WifiP2pManager and Channel data types are initialized. These variables are used in every block the Wifi Direct Service block encapsulates. The Channel data type includes a callback listener that is activated if the channel is lost. If this occurs, the *CHANNEL_LOST* event gets triggered. This will again lead to a triggering of the *failure* pin (see Fig. 5.13). On the first time the channel is lost, a re-initialization is attempted. If the channel is lost once again, the block assumes that the channel is permanently lost and gives up the initialization procedure.

The first of the API blocks to be initiated is the Wifi Direct Receive block. When this happens, the application will be able to receive Wi-Fi Direct related intents from the Android framework. Every time this happens, an update is sent out of the Wifi Direct Service block's *update* pin. Likewise, if the *notConnected* pin is triggered at the Wifi Direct Receive block, the Wifi Direct Service block's failure pin will be triggered.

When the *findAndConnect* pin is triggered, a variable with a WifiP2pConfig data type is conveyed into the block. This variable is needed to set up a Wi-Fi Direct connection. In order to keep this block as generic as possible, this variable must be configured on the outside of the block. Accordingly, the Wifi Direct Discover Peers block is initiated. This block will return a list of discovered devices. As a result, the *CONNECT* event is triggered and the Wifi Direct Connect block is initiated. This leads to a connection request being sent to the discovered devices. In the same way as with the Wifi Direct Receive block, these two blocks are able to trigger the failure pin at the Wifi Direct Service block if errors occur. In addition, the Wifi Direct Connect block can also trigger the Wifi Direct Service block's update pin when updates are conveyed out of the Wifi Direct Connect block.

If the *disconnect* pin at the Wifi Direct Service block is triggered, a disconnection procedure should be executed. However, this should only happen if a connection with a peer device is established. This is taken into account by having boolean variables declared in the Wifi Direct Service block.

A successful connection between the current device and a peer device can be accomplished even if the Wifi Direct Connect block hasn't been initiated. When this happens, the current device is the responding one and the peer device is the initiating one. In this case, triggering the disconnect pin at the Wifi Direct Connect

block is not possible because the block has never been initiated. Therefore, the Remove Group block also needs to be implemented outside the Wifi Direct Connect block. Finally, if the *stop* pin is triggered, all active blocks will be terminated along with the Wifi Direct Service block.

5.4.2 Analysis of the Wifi Direct Service block

Figure 5.14 shows the Wifi Direct Service block's ESM. It has four states besides the initial and final state, namely *listening, active, stopping* and *disconnecting*.

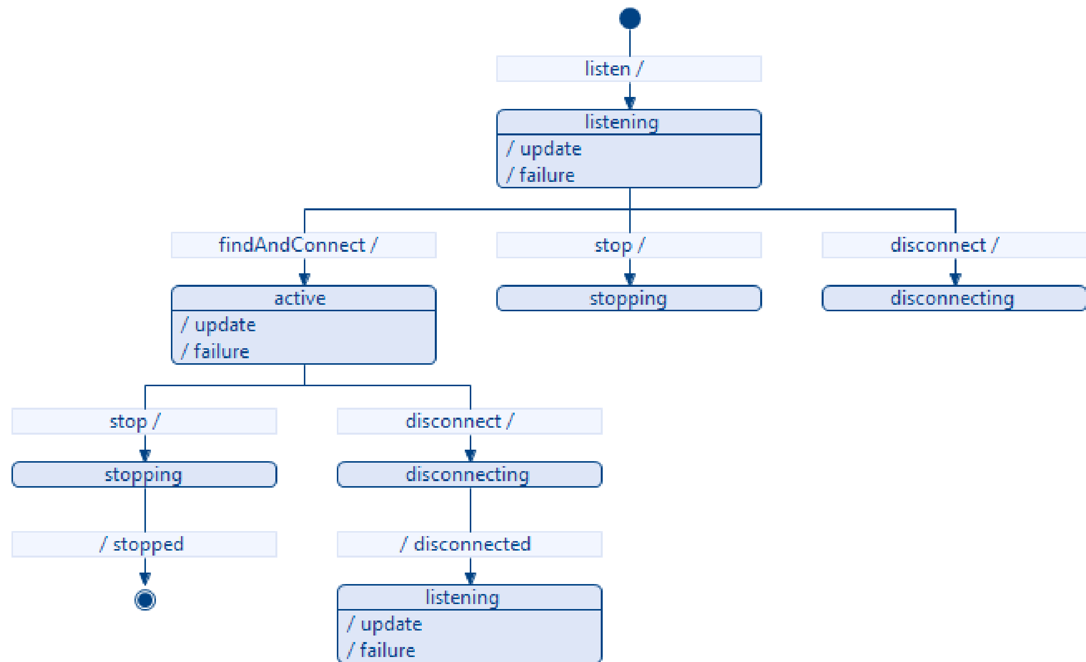


Figure 5.14: The ESM of the Wifi Direct Service block

When the listen pin is triggered, the block enters the listening state. In this state the following five pins can be triggered:

- update
- failure
- findAndConnect
- stop
- disconnect

A triggering of the update and failure pins will not cause the application to enter a new state, like the other three input pins do. The stop pin will cause the

application to enter the stopping state while the disconnect pin cause it enter the disconnecting state.

If the findAndConnect pin is triggered, the application will enter the active state where the discovering of peers and connection procedure are initiated. Since updates and error messages can occur while the application is in this state, the update and failure pins are still capable of being triggered.

When in disconnecting state, the application can only trigger the disconnected pin causing it to re-enter the listening state. Consequently, the application will again be able to discover and send connection requests to other peer devices. The only way to terminate the block is when it enters the stopping state and the output pin is triggered.

5.4.3 The Wifi Direct Responder and the Wifi Direct Initiator blocks

The Wifi Direct Service block restricts the sequence of how the Wi-Fi Direct API blocks are composed by encapsulating them. However, as an experiment we wanted to further specialize this block in order to accommodate even more specific use-cases. Therefore, we divided the Wifi Direct Service block's functioning in two separate use-cases. This was realized by composing two additional blocks, namely the *Wifi Direct Responder* and the *Wifi Direct Initiatior*, where each of them encapsulates the Wifi Direct Service block.

Figure 5.15 and Fig. 5.16 show the internal structure of these blocks. The Wifi Direct Responder block is responsible for informing the application when connections has been successfully initiated by a peer device. In addition, to keep the application updated with the status of the current device. The Wifi Direct Initiator block is responsible for letting the users find and initiate connection requests to peer devices. As the figures show, they differ slightly internally. The Wifi Direct Responder will not trigger the *findAndConnect* pin at the Wifi Direct Service block, whereas the Wifi Direct Initiator will. This is done after a short break to ensure that the *listen* pin is triggered first, in order to initiate the Wifi Direct Service block. None of these blocks has the opportunity to initiate a disconnection without being terminated. Hence, the *disconnect* and the *disconnected* pins at the Wifi Direct Service block are not used in these blocks.

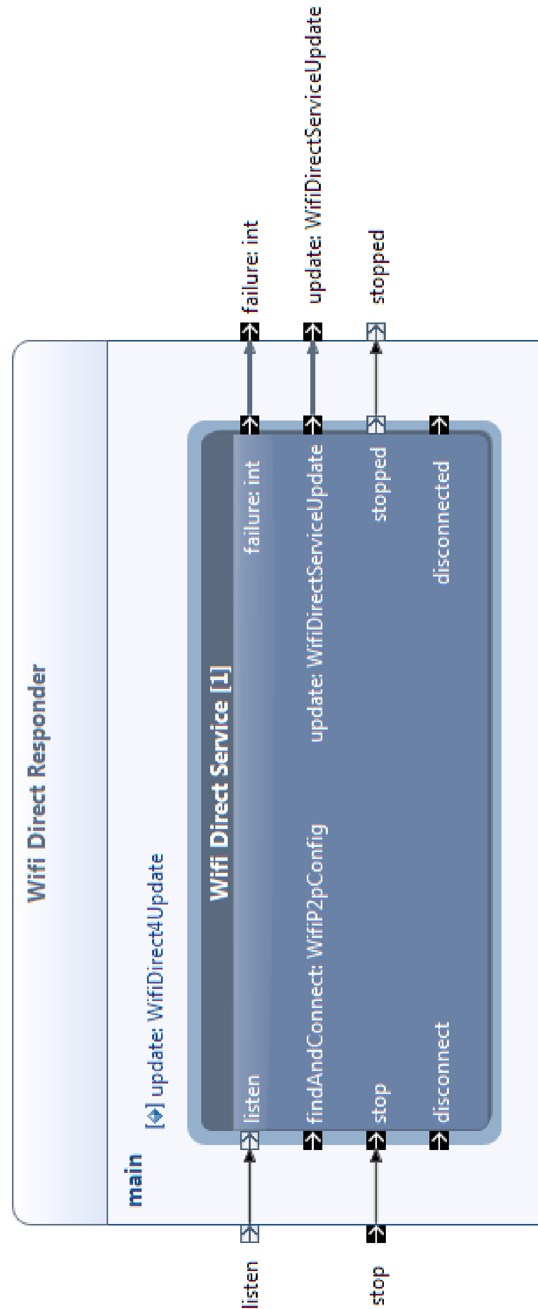
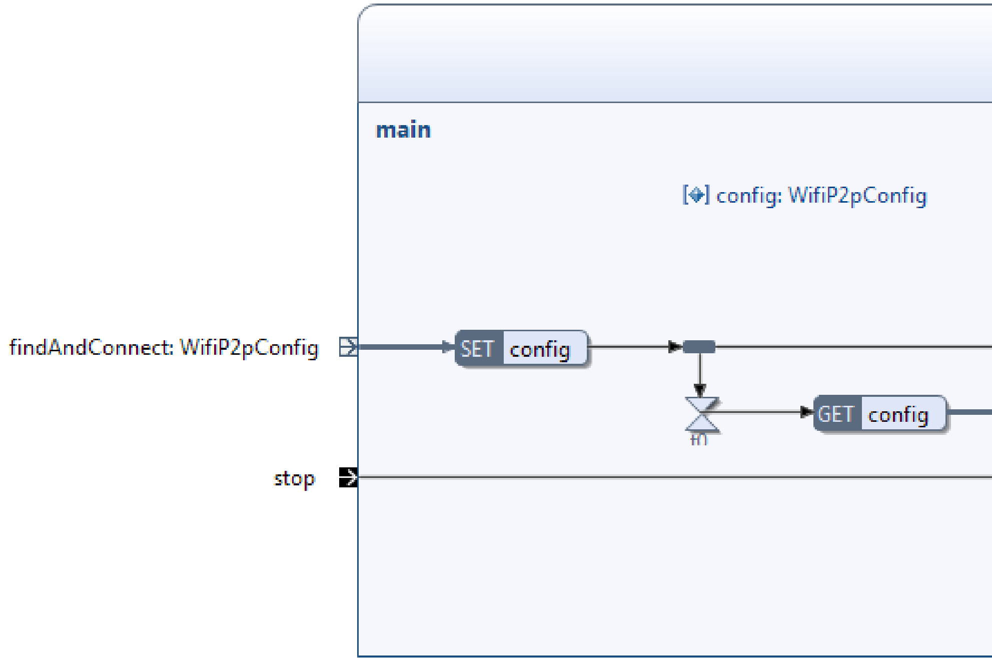


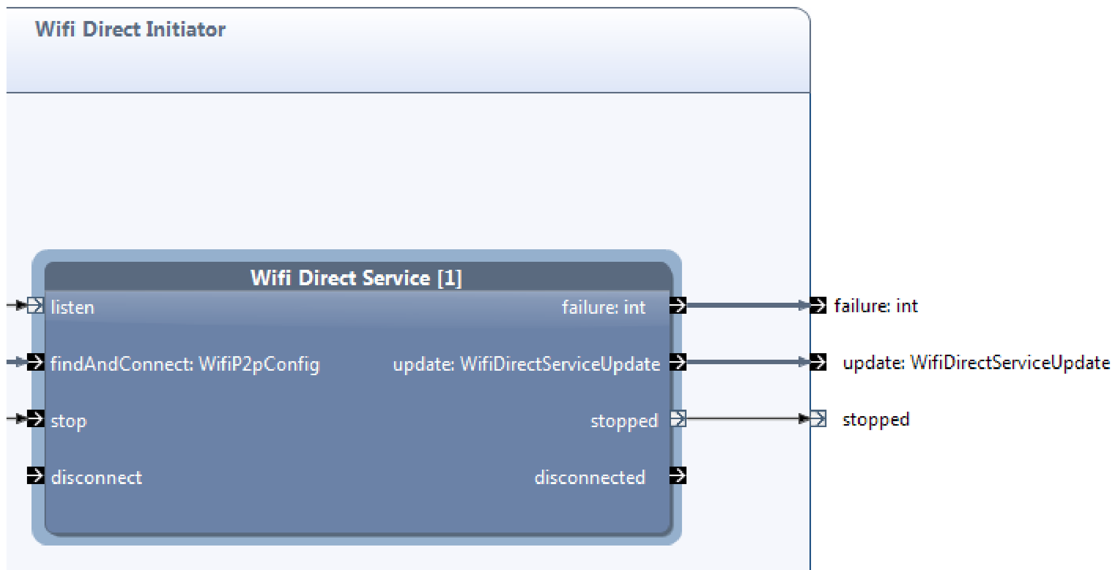
Figure 5.15: The Wifi Direct Responder block

5.5 Evolvement of the Wifi Direct Service block

All building blocks described so far are evolved by an iterative process between testing and restructuring, in order for them to meet their functional requirements. This section describes the progression of how the *Wifi Direct Service* block evolved, in order to give the reader an insight of this process.



(a) The left side of the block



(b) The right side of the block

Figure 5.16: The Wifi Direct Initiator block

5.5.1 The Wifi Direct 1 block

An entire new building block, called *Wifi Direct 1* was created in order to encapsulate the *Wifi Direct Receive*, *Wifi Direct Discover Peers* and *Wifi Direct Connect* blocks. The main objective of this block was to facilitate implementation of the Wi-Fi Direct service without profound knowledge of the Wi-Fi Direct technology

or Android's Wi-Fi Direct API.

At this moment, we noticed that the Android's Wi-Fi Direct API was quite closely connected to the Android's activity lifecycle. Therefore we started to create input pins identified in accordance to the various states of this lifecycle. Figure 5.17 shows these input pins.

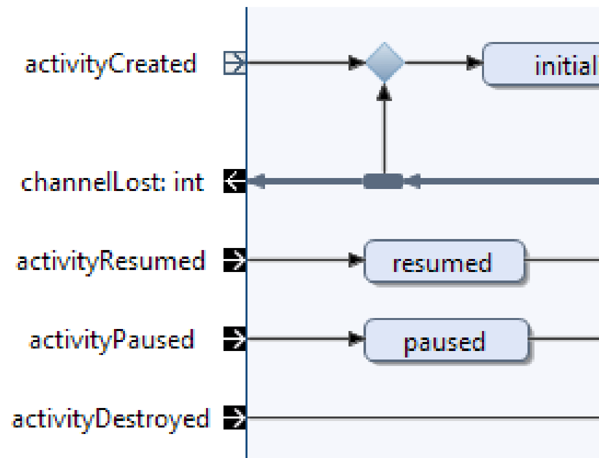


Figure 5.17: A snippet of some of the Wifi Direct 1 block's input pins

The purpose of this was to make it easier for developers to know when the various input pins should be triggered. However, the drawback was that the number of input pins hadn't been reduces compared to the number of input pins at the API blocks. This caused the block to continuously being overly complex. Despite this, we managed to reduce the total amount of pins from twenty-four to fourteen at this point in time. One of the reasons for this was that we assembled all error messages into one token flow instead of having a separate flow for each of them (see Fig. 5.18). By doing so, seven different output pins from the API blocks was concatenated down to one single pin called *failure* at the Wifi Direct 1 block.

In addition, both the Wifi Direct Receive and the Wifi Direct Connect block have some identical output pins, respectively *connectionInfo* and *groupInfo*. These four pins are combined into two token flows, one for each pin type. Furthermore, the Wifi Direct 1 block separates whether the current device is a group owner or a client (see Fig. 5.19). This is determined by whether the Wifi Direct Receive or Wifi Direct Connect block's connectionInfo pin that was triggered. As a consequence, these four pins are only reduced by one.

Figure 5.18 also shows that the token flow out of the *peersAvailable* pin at the Discover block (this block is essentially the same as the Wifi Direct Discover Peers block) is directly connected to the *connect* pin at the Wifi Direct Connect block.

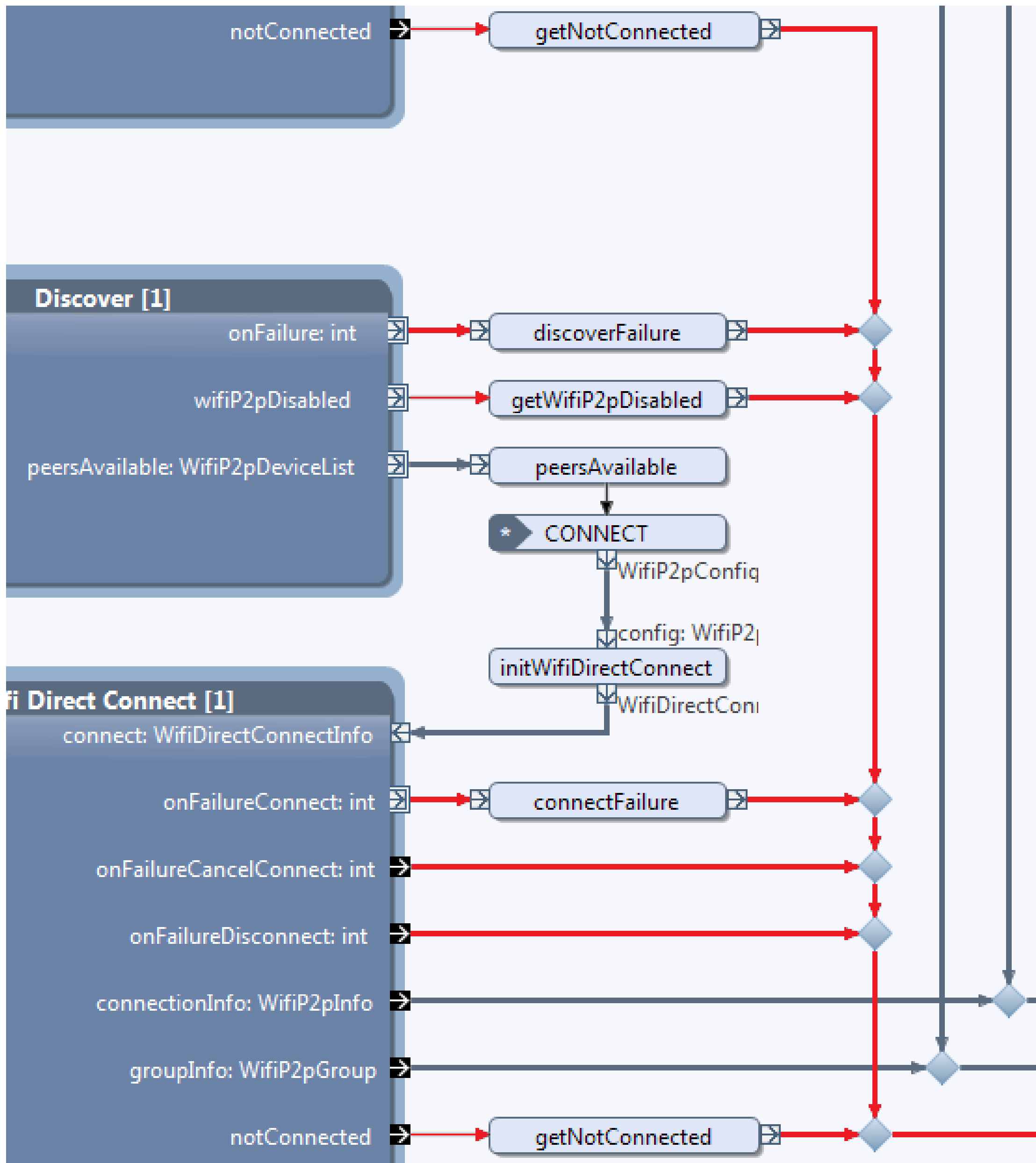


Figure 5.18: A snippet of the Wifi Direct 1 block. The error message flows are emphasized by a red color

Hence by using the Wifi Direct 1 block, the user will not be able to manually initiate a connection to a peer device. This result in fewer pins to deal with, which again reduces the block's complexity.

5.5.2 The Wifi Direct 3 block

After some additional experiments and analysis, we got another idea on how to further reduce the amount of input pins. Therefore, we duplicated the Wifi Direct 1 block into a new block named *Wifi Direct 3* to try a new approach.

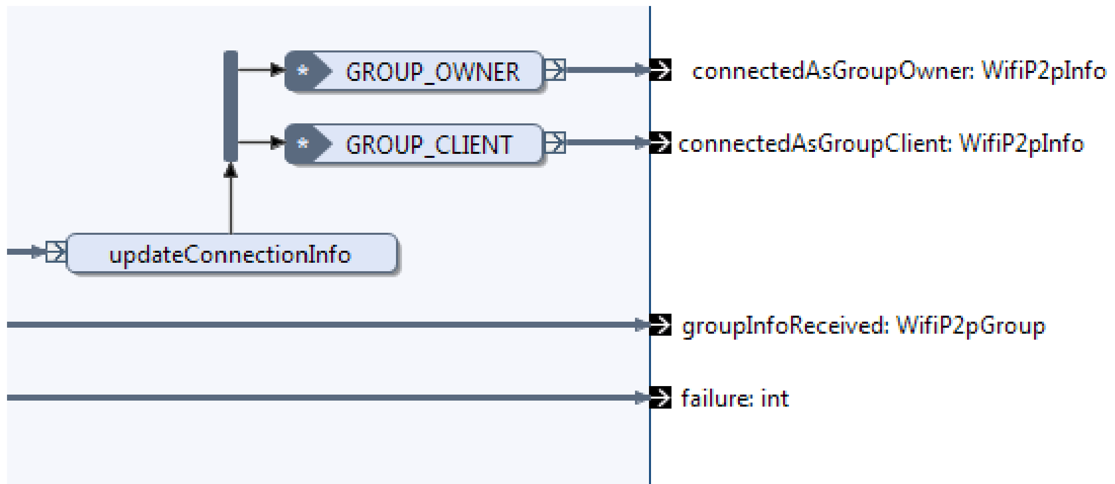


Figure 5.19: A snippet of the Wifi Direct 1 block

In this block we discarded the idea of having the input pins corresponding to the Android's activity lifecycle. Instead, we made them consistent to the users' interaction with the application. As a consequence, the total number of input pins was reduced from seven to four. This is shown in Fig. 5.20.

5.5.3 The Wifi Direct 4 block

Finally, we wanted to combine all the informative updates the various API blocks are listening for, in the same manner as we did with the error messages in the Wifi Direct 1 and Wifi Direct 3 block. Therefore, we duplicated the Wifi Direct 3 block into a new block named *Wifi Direct 4* to include this additional behavior. This is shown in Fig. 5.21. The informative updates will trigger the following output pins at the Wifi Direct Receive and the Wifi Direct Connect blocks:

- thisDeviceChanged
- connectionInfo
- groupInfo

If any of the latter pins are triggered, the Wifi Direct 4 block will check whether the exact same update has previously been conveyed out of the block. If not, the *update* pin at the Wifi Direct 4 block gets triggered.

Some additional logic was added to the building block after some further testing. As a result, we ended up with the Wifi Direct Service block described in Sect. 5.4 (see Fig. 5.11, Fig. 5.12 and Fig. 5.12).

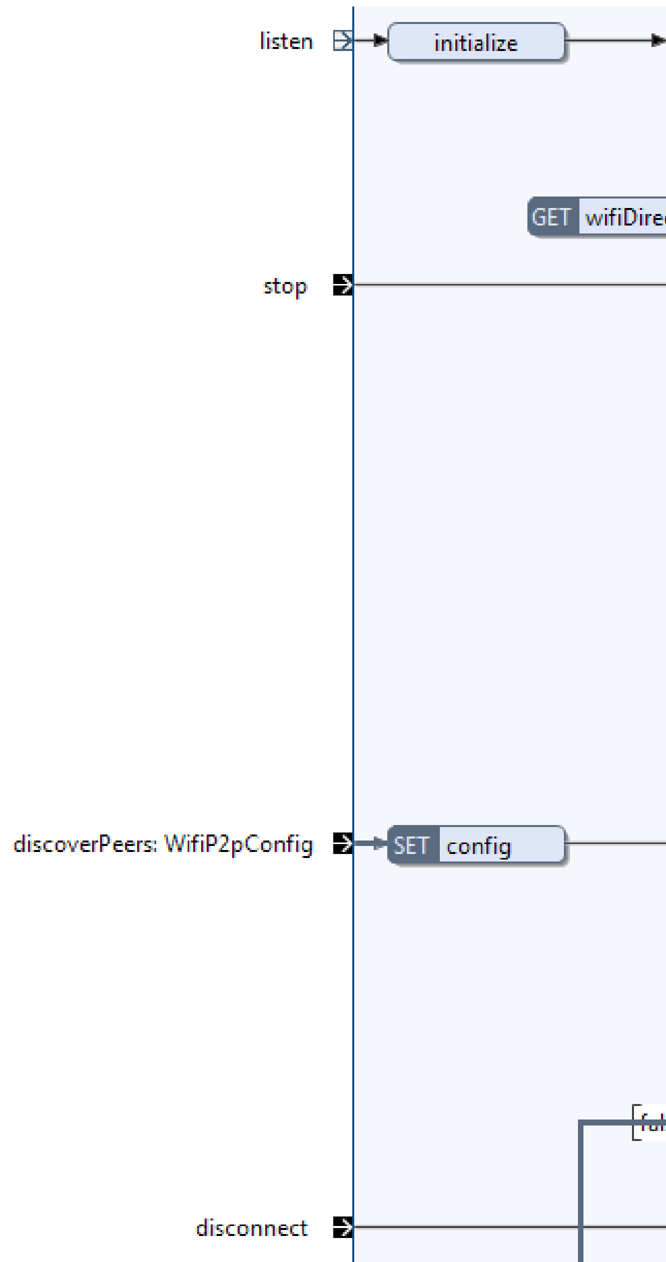


Figure 5.20: A snippet of the Wifi Direct 3 block's input pins

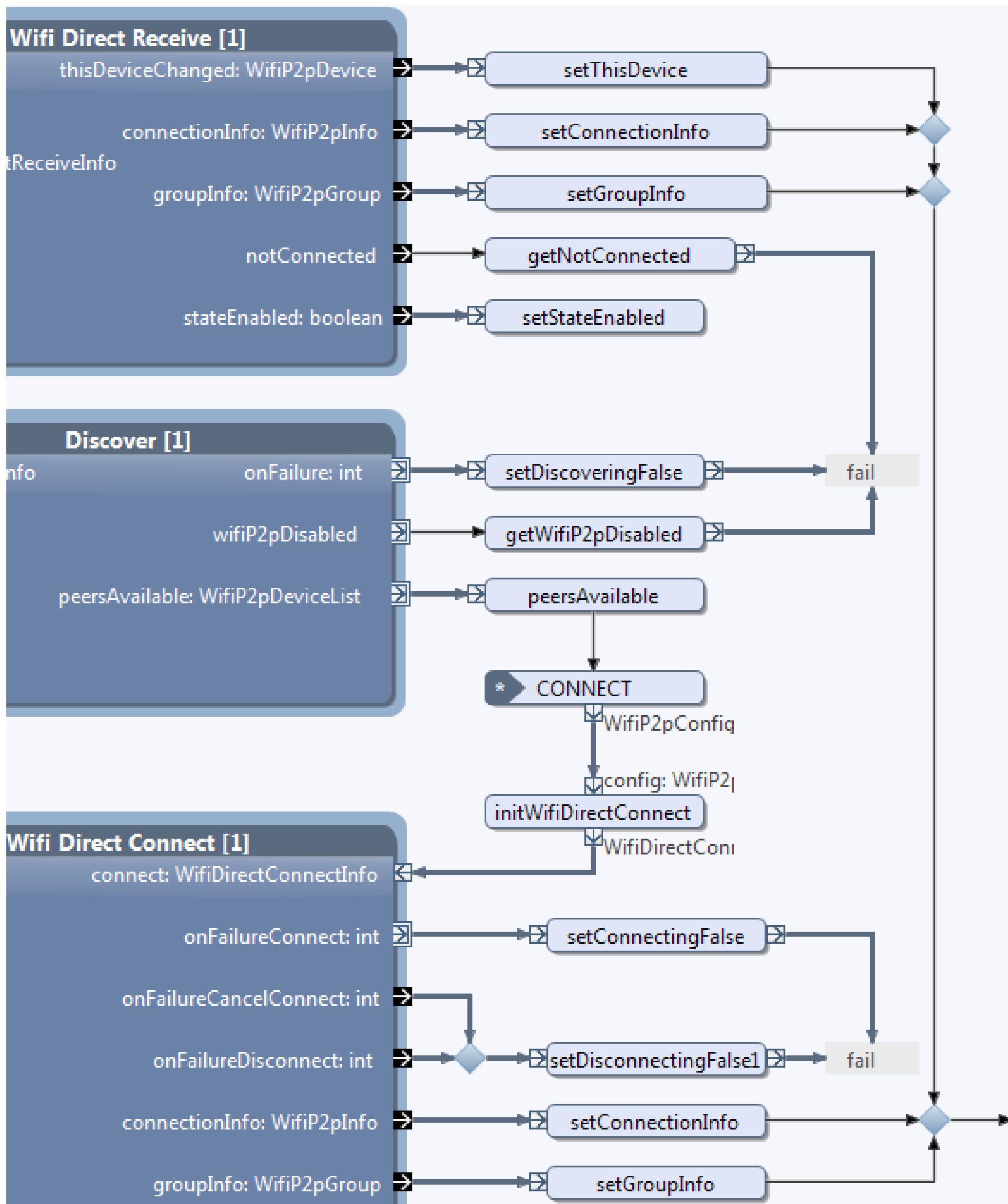


Figure 5.21: A snippet of the Wifi Direct 4 block

Chapter 6

Example Application

To demonstrate how the API building blocks from Chapt. 5 can be used, we developed an application with the following functional requirements:

- It should be able to receive and react to connection requests from other devices by using Wi-Fi Direct.
- It should be able to discover other peer devices by using Wi-Fi Direct.
- It should be able to initiate connection requests to discovered peer devices by using Wi-Fi Direct.
- Once a connection with a peer device is initiated, it should be able to receive P2P group information and display it to the users via its UI.
- The same application should be able to act both as a group owner and a client.
- It should be able to initiate disconnection requests when connected to a group.

The following functional requirement should be specific for client devices:

- It should be able to take a photo on command from the group owner and automatically send it to the group owner device.

The following functional requirements should be specific for group owner devices:

- It should be able to open and close the client device's camera.

- When the camera is opened, should be able to choose which camera the client should use, either the front or the back camera.
- It should be able to command the client device to take a photo.
- Once a photo is received from the client device, it should automatically save and display it on the mobile device.

6.1 Application building blocks

Figure 6.1 shows the *Wifi Direct System* block. This block is at the highest decomposition level of the application. It contains two inner blocks, namely *Startup* and *Wifi Direct Application Overview*. The Startup block is used to aid an Android application to start, terminate and re-activate. The other block encloses the whole application logic, from an activity being created until it is destroyed.

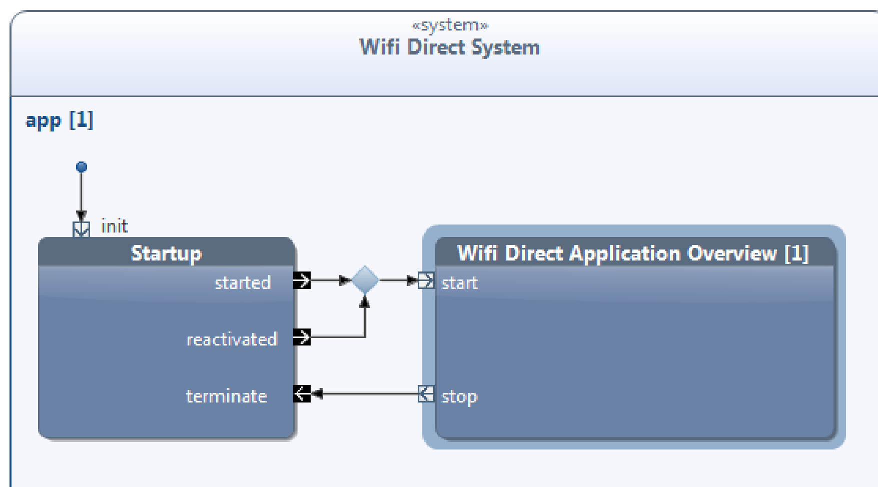


Figure 6.1: The Wifi Direct System block

6.1.1 The Wifi Direct Application Overview block

Figure 6.2, Fig. 6.3 and Fig. 6.4 shows the internal structure of the Wifi Direct Application Overview block. The first building block to be initiated inside this block is a block named *Activity* (see Fig. 6.2). This block starts an Android activity and listens to its lifecycle. Its activity class is identified as *WiFiDirectApplicationActivity*. This class can trigger the following seven events in the Wifi Direct Application Overview block:

- *DISCONNECTED*
- *ENABLE_SELECTED*

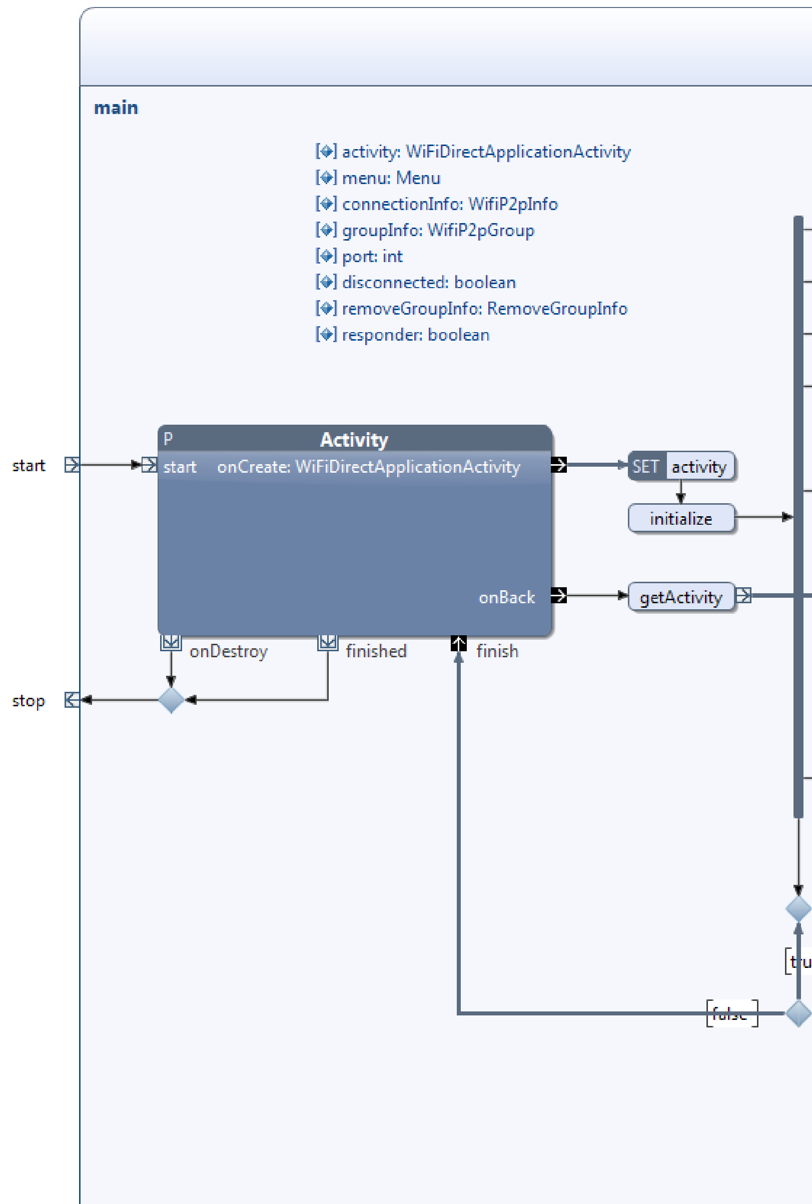


Figure 6.2: The left side of the Wifi Direct Application Overview block

- *ON_CREATE_OPTIONS_MENU*
- *DISCOVER_SELECTED*
- *SEND_PHOTO*
- *GROUP_PHOTO*
- *BACK*

When the Activity block is initiated, the *Wifi Direct Responder* and the *Remote Camera Service* block will be triggered (see Fig. 6.3 and Fig. 6.4). The Wifi Direct

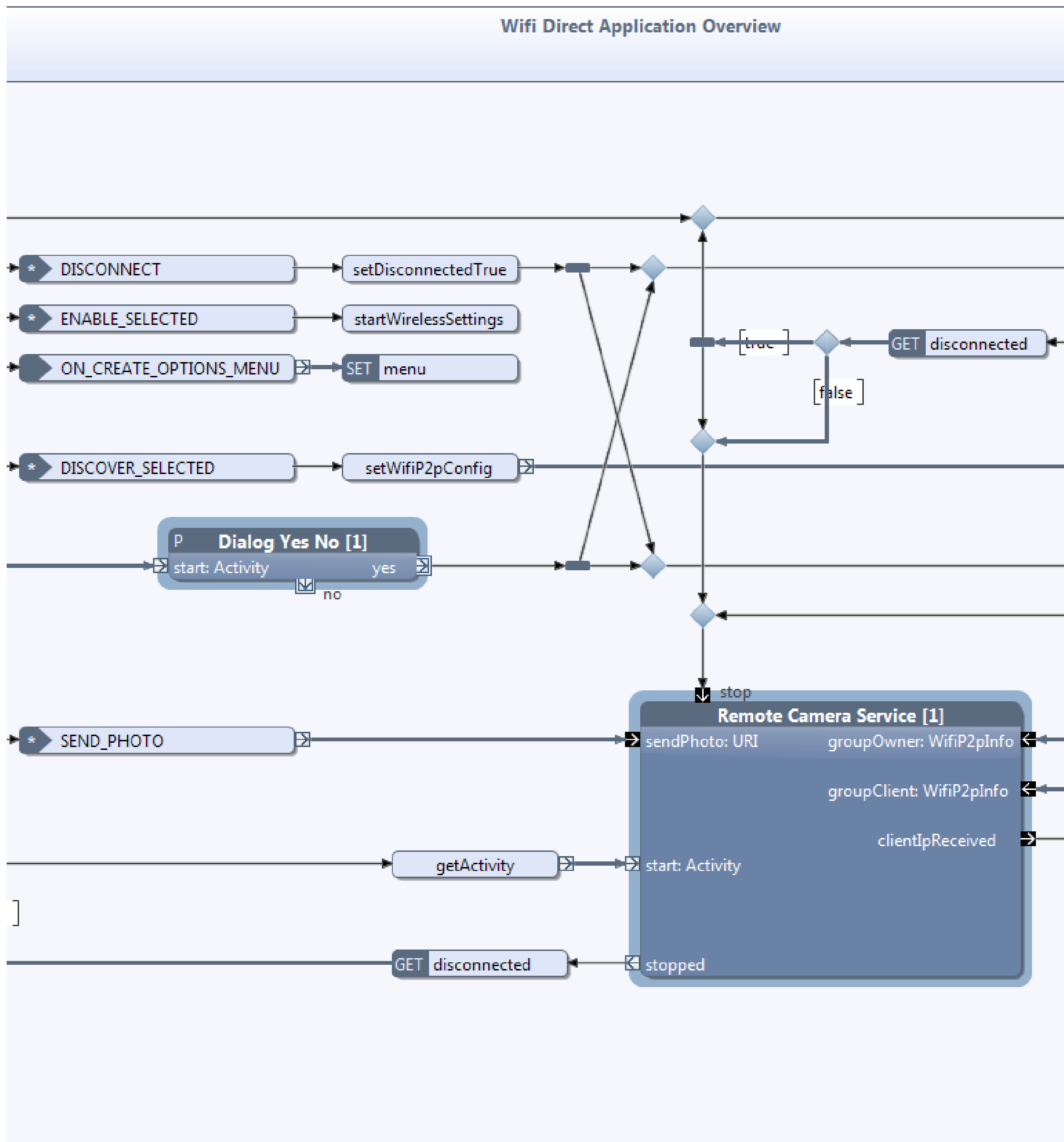


Figure 6.3: The middle part of the Wifi Direct Application Overview block

Responder block is responsible for listen to connection request from peer devices, while the Remote Camera Service block is responsible for the TCP transport between the devices. A capture of the application's UI at this stage is shown in Fig. 6.5.

The DISCOVER_SELECTED event in Fig. 6.3 gets triggered when the user pushes the discover peers button at the UI (the magnifying glass icon at the top-right corner in Fig. 6.5). This results in an initiation of the *Wifi Direct Initiator* block. There are two possible outcome of this action:

- The discover peers initiation fails and an error message is given to the users,

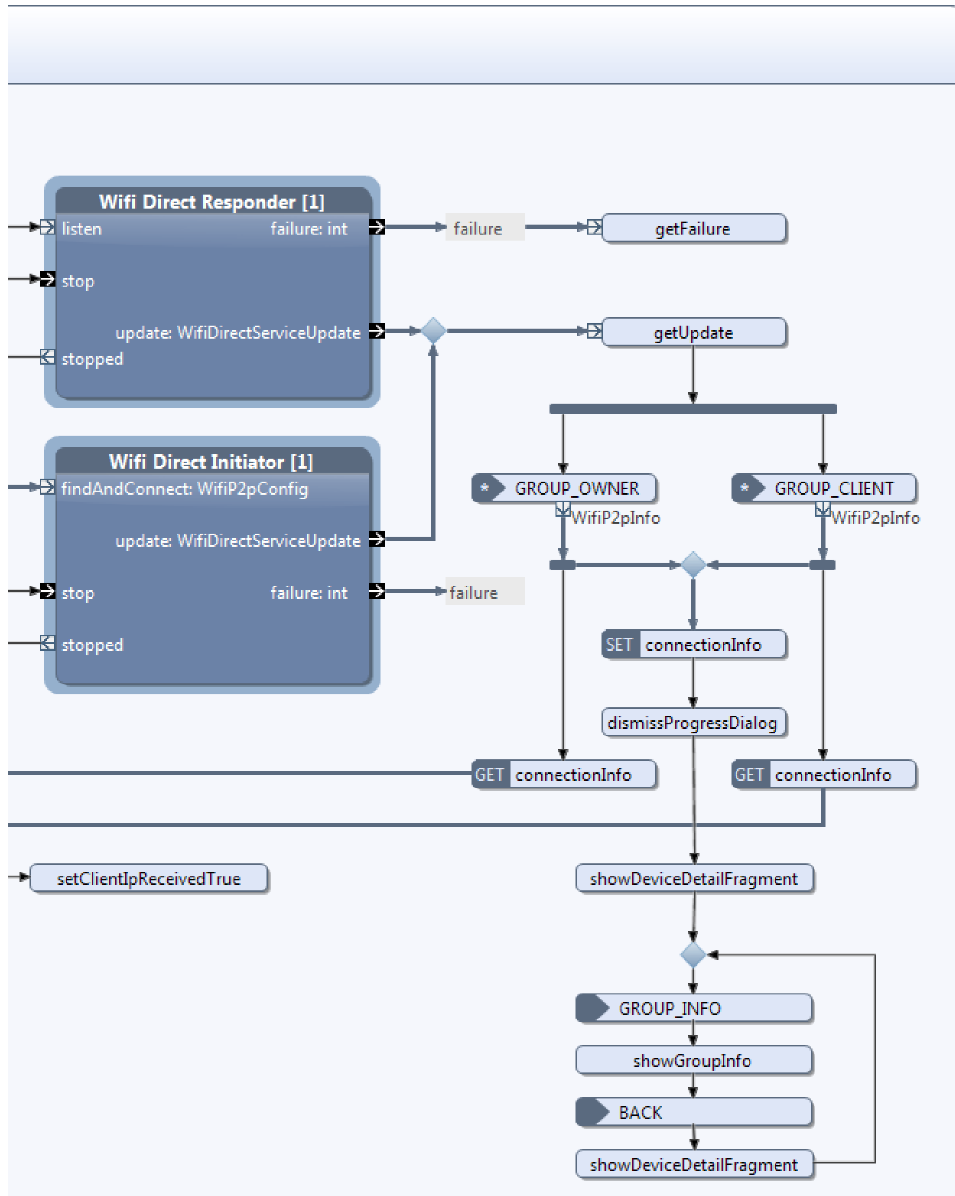


Figure 6.4: The right side of the Wifi Direct Application Overview block

e.g. the Wi-Fi Direct mode is turned off (see Fig. 6.6a).

- The discover peers initiation succeeds and a scan for peer devices starts. The magnifying glass is changed with a progress bar to indicate that the scan has started (see Fig. 6.6b).

If the discover peer initiation process succeeds and a peer device is discovered, a connection request is automatically sent to the peer device causing an alert dialog to be displayed on the peer device's UI (see Fig. 6.7).

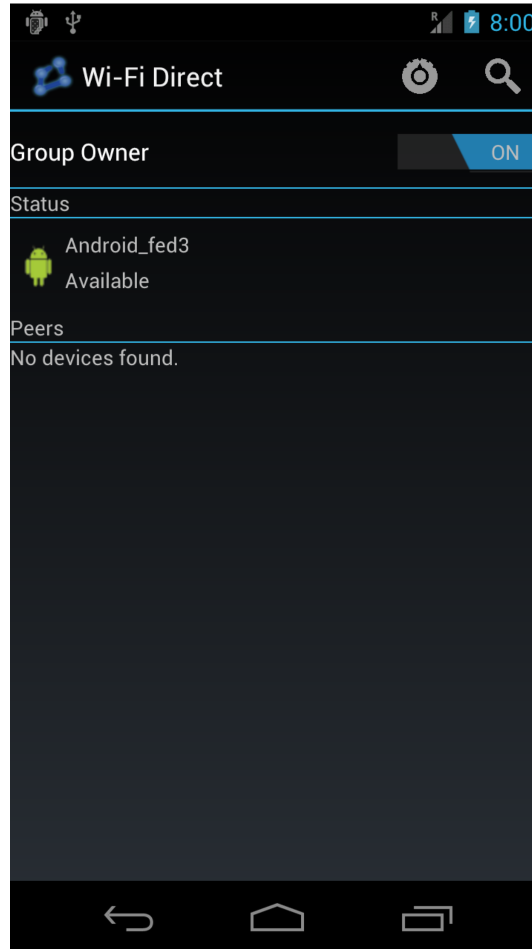
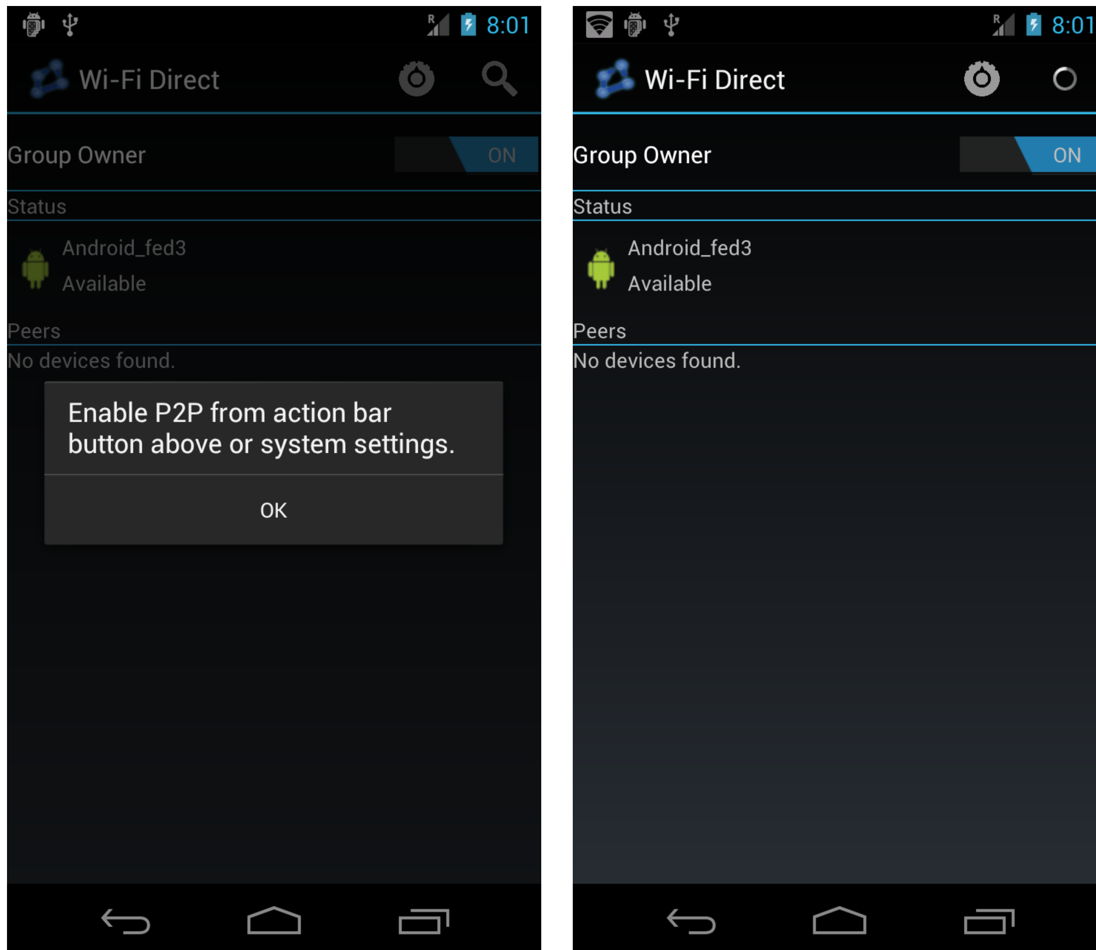


Figure 6.5: The application is initialized

If the peer device accepts the connection request, a group negotiation to determine the group owner is started. If the initiating device has turned the *Group Owner* switch to *ON* (see Fig. 6.5), the current device will become the group owner. Otherwise, the peer device becomes the group owner. See Sect. 4.2 for more information of the group negotiation process.

If the group negotiation succeeds, the devices will be connected. If the current device is the group owner, the *GROUP_OWNER* event in Fig. 6.4 is triggered causing the Remote Camera Service block's *groupOwner* pin to be triggered. Otherwise, the *GROUP_CLIENT* event and subsequently the *groupClient* pin at the Remote Camera Service block is triggered. Captures of the UI at this stage is shown in Fig. 6.8 where Fig. 6.8a shows the group owner's UI and Fig. 6.8b shows the client's UI. These figures show that both devices have two buttons, respectively named *Disconnect* and *Group Info*. The Disconnect button triggers the DISCONNECT event in Fig. 6.3 to initiate a disconnection procedure, which means that the current device gets removed from the group. By pressing the Group Info button,



(a) The discovery initiation process has failed

(b) The discovery initiation process has succeeded

Figure 6.6: The discovery initiation process with two possible outcomes

the GROUP_INFO event in Fig. 6.4 is triggered and the UI will display information about the group. This is shown in Fig. 6.9 where Fig. 6.9a shows the group owner's UI and Fig. 6.9b shows the client's UI. Notice from this figure that the group owner will hold group's passphrase.

In addition to the Disconnect and Group Info button, the group owner device has a button named *Open Camera* (see Fig. 6.8a). When this button is pushed, a message will be sent to the client device instructing it to open the camera. The client device's UI at this point in time is shown in Fig. 6.10b. Accordingly, an additional button named *Take Photo*, in addition to a switch is shown at the group owner's UI (see Fig. 6.10a). In addition, the Open Camera button is changed to *Close Camera* in order to reverse this process.

When the Take Photo button is pushed, a message is sent to the client device instructing it to take a photo and transfer it back to the group owner device. The

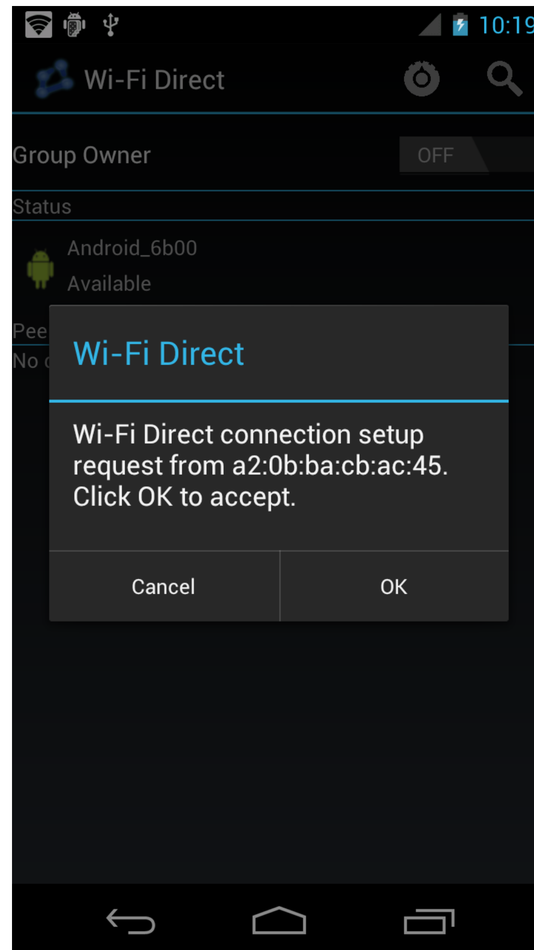
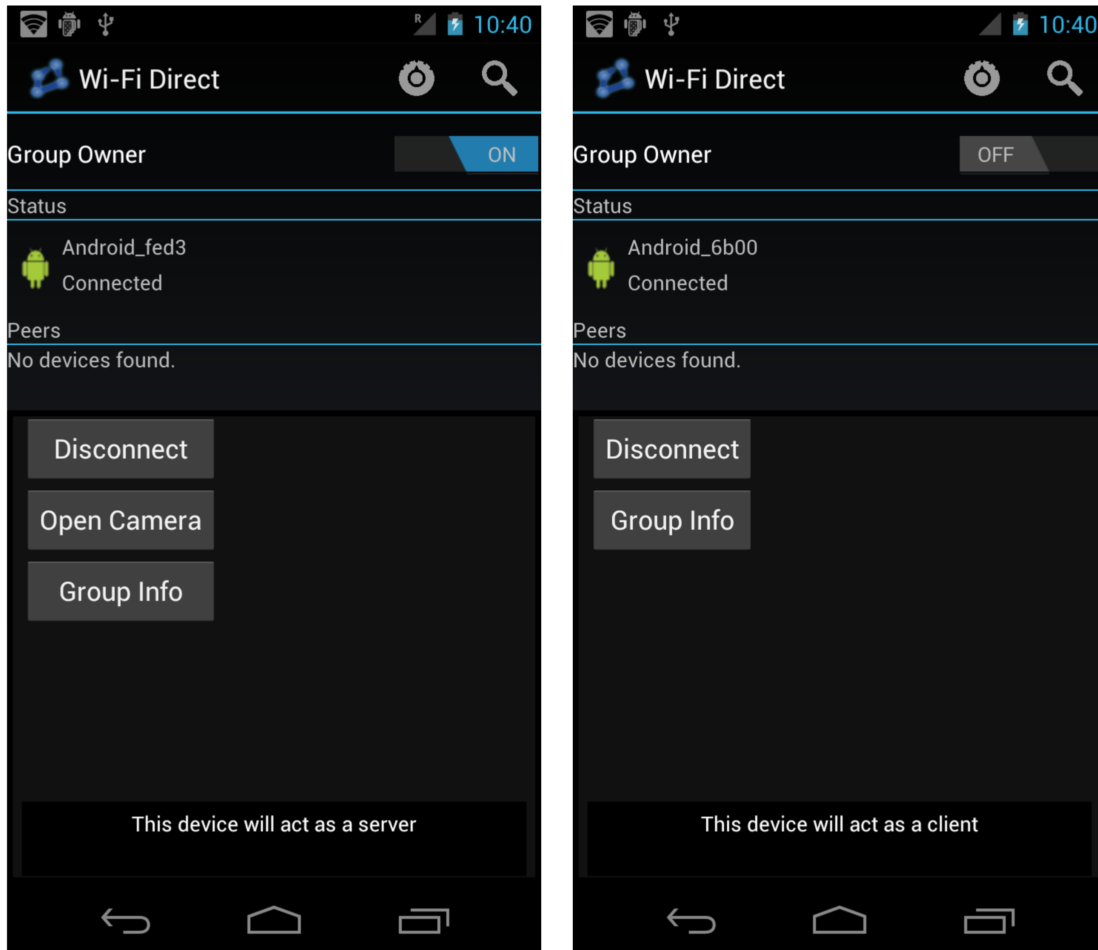


Figure 6.7: A connection request is received

captured photo is subsequently saved and displayed on the group owner device.

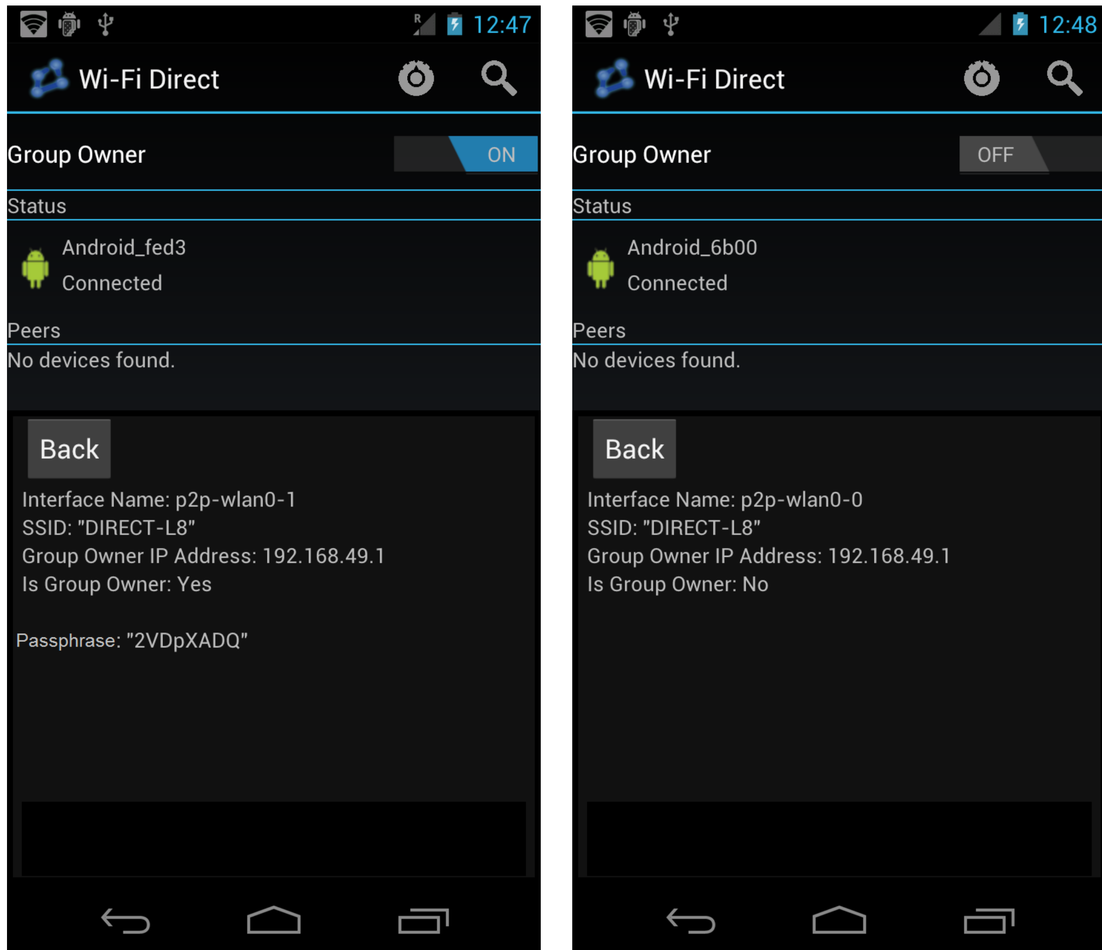
The switch controls which camera the client device is going to use, either the front or the back camera. Every time the switch is alternated a message is sent to the client device instructing it to switch the camera.



(a) The group owner device

(b) The client device

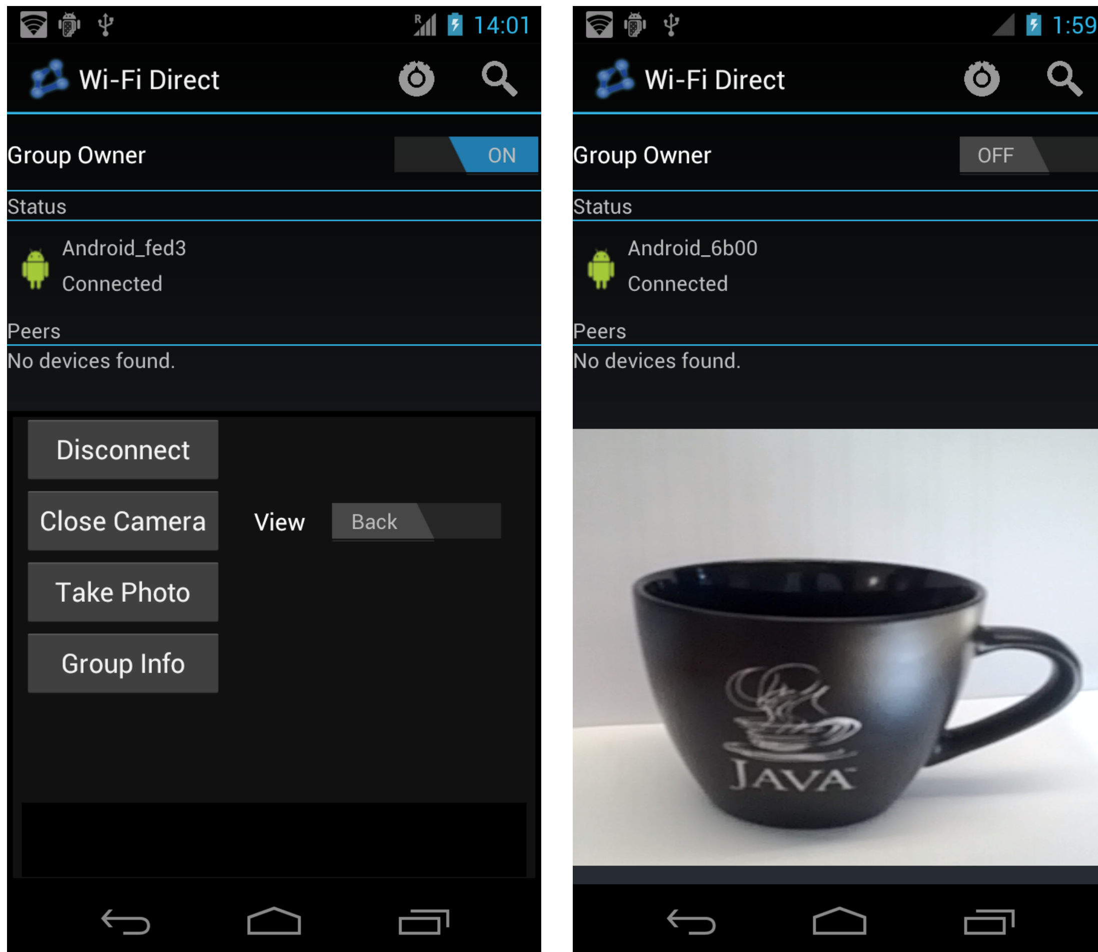
Figure 6.8: The devices are connected



(a) The group owner device

(b) The client device

Figure 6.9: Group information is displayed on the UI



(a) The group owner device (b) The client device

Figure 6.10: The Open Camera button has been pushed on the group owner device's UI

Chapter 7

Discussion

This chapter describes the results of the study. The importance of the findings is pointed out and compared with prevailing approaches of software development. In addition, the limitations that were exposed during the study are called attention to. Finally, some suggestions for further work are emphasized in order to further endorse the results.

7.1 Findings

Our study confirms that using Arctis building blocks based on an API will enhance the understanding of the API's intended sequence of events. This is done by graphically visualize the course of events in an editor instead of only having it written down in an API documentation. It is much easier to follow a building block's token flow than examining the code of an application in order to understand the sequence of events. In addition, as pointed out in Sect. 3.2, Arctis automatically analyzes the model for design flaws as the blocks' token flow are manually followed.

The ordinary way of implementing functionalities by using an API can result in unnecessary communication between the software components. This will result in a reduced efficiency and a lack of responsiveness in the application. Since mobile devices have a limited amount of resources, the Android system has established a protection against unresponsive application. Each time an application doesn't meet the requirements of responsiveness, the system will display an Application Not Responding (ANR) dialog (see Fig. 7.1) [26]. This dialog leads to an interrupted user experience and conducts developers to design responsive applications.

Much developing time is saved by directly applying a building block with a pre-

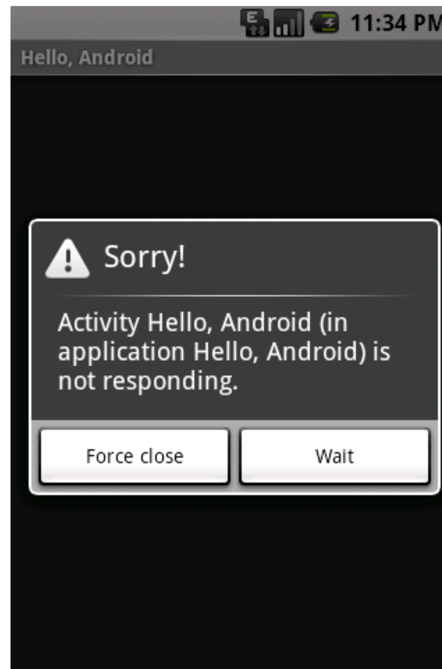


Figure 7.1: Example of an ANR dialog. Taken from [26]

defined behavior into an application. By doing so, developers don't need to spend additional time on gaining in-depth knowledge of the API in order to identify how the sequence of events should be, and which API methods that are blocking. As long as they know what kind of use-cases a specific API block ought to be used in and when a building block's pins should be triggered, the block may easily be implemented. In addition to saved implementation time, the testing time will be significantly reduced because the API blocks should already be thoroughly tested in order to function with a set of predefined use-cases.

By using API building blocks, a complex behavior can be decomposed into a hierarchy of sub-blocks affecting developers to abstract themselves from unnecessary details in order to comprehend the overall behavior. Hence, the probability of having errors arise upon implementation can be further reduced with this approach.

Another aspect of using the API blocks is the portability of functionality. These building blocks are made to be portable, i.e. they are made to be implemented in other developers' applications. On the other hand, developers should not expect that a sequence of events written in code can seamlessly be relocated from one generic use-case into specific scenarios within other applications. Building blocks restrict developers from operating at the code level, which enhances the separation of functionalities. It would for instance be a relatively easy task to either remove or change an API block. This is because the functionality is confined within

the block. On the other hand, by removing or altering some code without using building blocks may result in unexpected effects on other functionalities.

7.2 Limitations

Occasionally, the Android APIs will have some of their functions updated or deprecated. However, modifications of Android's APIs will not affect Arctis building blocks based on them. Thus, these blocks must be brought up to date manually. As a consequence, the API building blocks can be outdated without developers necessary being aware of it.

A fundamental requirement for implementing the building blocks into an application is that developers have sufficient knowledge of how to use the Arctis tool. Even though using it is quite intuitive, there are some concepts developers should acquire before beginning to model and implement building blocks.

Using a building block in order to embrace functionality into an application is difficult and time-consuming if the application hasn't been using the Arctis tool from the beginning. In case only a minor functionality is going to be implemented, the time gained on implementing the building block will probably be lost if the whole application's behavior needs to be replaced by Arctis models.

7.3 Further Work

In this thesis, some aspects on how Wi-Fi Direct could be implemented in a system have been discussed. However, in order to completely cover the Android's Wi-Fi Direct API, it still remains some use-cases to be explored and discussed. In addition, building blocks from other APIs could be created in order to demonstrate that this method of developing is not constricted to a single API.

In order to examine the actual time saved by using the API blocks, we could measure the time some selected developers spend on implementing a specific functionality with a set of predefined API blocks. Furthermore, we could compare this time to the time some other developers spend on implementing the exact same functionality, but without using the API blocks.

Chapter 8

Conclusion

Wi-Fi Direct is one of many technologies which has recently made its entrance to the Android platform. In order to utilize and implement such up-to-date technologies, developers need to gain sufficient knowledge. This is sometimes challenging and much time is therefore spent on understanding them. By using Arctis SDK, we have developed building blocks to ease the interpretation of this functionality.

The building blocks emerged through an iterative process between establishing the requirements, analysis and design, implementation and evaluation. Together with Pixavi, a set of use-cases has been devised to verify the blocks requirements. In order to prove the utility of the blocks, they were implemented in an example application. This application was further tested and evaluated before the requirements once again were examined. This iterative process continued until the blocks covered a satisfactory amount of functionality according to the requirements.

A significant amount of development time will be reduced by using the building blocks we have presented in this thesis. This is due to the following reasons:

- Using a graphical notation to present the expected behavior will enhance the understanding. As a result, the time developers need to spend in to gain sufficient amount of knowledge will decrease.
- The blocks add contracts to the behavior, which makes it less prone to design flaws. Thus, the testing and error correction time will be reduced.
- The implementations of functionality from the Android's Wi-Fi Direct API are already completed within these blocks.

By using these results as a foundation, further development of building blocks can be realized by exploring more APIs and use-cases. It is hoped that this way of thinking will inspire other developers to see the value of incorporating graphical models in the development process.

References

[1] Google Inc. and Open Handset Alliance. *WifiP2pManager*. [Online]. Available: http://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.html. [Accessed February 28, 2012].

[2] David D. Coleman and David A. Westcott. *CWNA: Certified Wireless Network Administrator Official Study Guide: Exam PW0-104*. John Wiley & Sons, 2009.

[3] Wi-Fi Alliance. Wi-fi certified wi-fi direct: Personal, portable wi-fi technology. October 2010.

[4] Wi-Fi Alliance. Wi-fi gets personal: Groundbreaking wi-fi direct launches today, October 2010. [Online]. Available: http://www.wi-fi.org/media/press-releases/wi-fi%C2%AE-gets-personal-groundbreaking-wi-fi-direct%E2%84%A2-launches-today. [Accessed February 23, 2012].

[5] Wi-Fi Alliance. Wi-fi certified wi-fi protected setup: Easing the user experience for home and small office wi-fi networks. December 2010.

[6] Olli Vihervuori. Recent developments in ieee 802.11 wireless local area network link-layer security. April 2009. [Online]. Available: http://cse.tkk.fi/en/publications/B/5/papers/vihervuori_final.pdf. [Accessed May 22, 2012].

[7] Daniel Camps-Mur, Xavier Pérez-Costa, and Sebastià Sallent-Ribes. Designing energy efficient access points with wi-fi direct. *Computer Networks*, 2011. [Online]. Available: http://www.campsmur.cat/files/wifi_direct_CN.pdf. [Accessed May

21, 2012].

[8] Google Inc. and Open Handset Alliance. *Android 4.0 Platform Highlights*. [Online]. Available: http://developer.android.com/sdk/android-4.0-highlights.html. [Accessed February 24, 2012].

[9] Google Inc. and Open Handset Alliance. *Platform Versions*. [Online]. Available: http://developer.android.com/resources/dashboard/platform-versions.html. [Accessed May 21, 2012].

[10] Google Inc. and Open Handset Alliance. *Intents and Intent Filters*. [Online]. Available: http://developer.android.com/guide/topics/intents/intents-filters.html. [Accessed May 7, 2012].

[11] Google Inc. and Open Handset Alliance. *Application Fundamentals*. [Online]. Available: http://developer.android.com/guide/topics/fundamentals.html. [Accessed May 8, 2012].

[12] Google Inc. and Open Handset Alliance. *Wi-Fi Direct*. [Online]. Available: http://developer.android.com/guide/topics/wireless/wifip2p.html. [Accessed May 12, 2012].

[13] Google Inc. and Open Handset Alliance. *WifiP2pManager.ChannelListener*. [Online]. Available: http://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.ChannelListener.html. [Accessed May 12, 2012].

[14] Google Inc. and Open Handset Alliance. *The AndroidManifest.xml File*. [Online]. Available: http://developer.android.com/guide/topics/manifest/manifest-intro.html. [Accessed May 11, 2012].

[15] Google Inc. and Open Handset Alliance. *android.net.wifi.p2p*. [Online]. Available: http://developer.android.com/reference/android/net/wifi/p2p/package-summary.html. [Accessed May 11, 2012].

[16] Google Inc. and Open Handset Alliance. *Manifest.permission*. [Online]. Available: http://developer.android.com/reference/android/Manifest.permission.html. [Accessed May 11, 2012].

[17] Google Inc. and Open Handset Alliance. *<uses-feature>*. [Online].

Available: http://developer.android.com/guide/topics/manifest/uses-feature-element.html. [Accessed May 11, 2012].

[18] Bitreactive. [Online]. Available: http://www.bitreactive.com/technology/faq. [Accessed June 2, 2012].

[19] Henriette Baumann, Patrick Grässle, and Philippe Baumann. *UML 2.0 in Action*. Packt Publishing Ltd., 2005.

[20] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Pearson Education Inc., 2004.

[21] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[22] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Use Case Approach*. Addison-Wesley Proffesional, 2003.

[23] Rajib Mall. *Fundamentals of Software Engineering*. PHI Learning Private Limited, third edition, 2009.

[24] Agile vs. other software-development methods. [Online]. Available: http://pg-server.csc.ncsu.edu/mediawiki/index.php/CSC/ECE_517_Fall_2010/ch6_6d_NM. [Accessed June 13, 2012].

[25] Google Inc. and Open Handset Alliance. *logcat*. [Online]. Available: http://developer.android.com/guide/developing/tools/logcat.html. [Accessed June 1, 2012].

[26] Google Inc. and Open Handset Alliance. *Designing for Responsiveness*. [Online]. Available: http://developer.android.com/guide/practices/design/responsiveness.html. [Accessed June 1, 2012].